AD-A038 338   ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB   F/G 9/2
IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS.(U)
OCT 76   J H PATEL   DAAB07-72-C-0259
UNCLASSIFIED   R-747   NL

1 OF 3
AD
A038338

AD A038338

FB.

REPORT R-747 OCTOBER, 1976 UILU-ENG 76-2235

COORDINATED SCIENCE LABORATORY

12

IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS

JANEK H. PATEL

D D C
RECEIVED
APR 18 1977
D

APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED.

AD No. ___
DDC FILE COPY

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) 6 IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS,		5. TYPE OF REPORT & PERIOD COVERED 9 Technical Report,
		6. PERFORMING ORG. REPORT NUMBER 14 R-747, UILU-ENG-76-2235
7. AUTHOR(s) 10 Janek H. Patel		8. CONTRACT OR GRANT NUMBER(s) MCS 73-03488 A01 15 DAAB 07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NSF-MCS-73-03488
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE 11 October, 1976
		13. NUMBER OF PAGES 193
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) 12 205p.		15. SECURITY CLASS. (*of this report*) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Pipelined Computers
Buffered Systems
Parallel Processing

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

A pipeline as defined here is a collection of segments of hardware which can operate simultaneously. A task flows synchronously from segment to segment for its execution. Each task follows one of several distinct task flow patterns which are assumed to be fixed and known in advance.

It is characteristic of pipelines that a task can be initiated in the pipeline before an earlier initiated task has completed its execution. A problem arises when two or more tasks try to use the same segment at the same time, causing a collision. Such collisions can be avoided by appropriate scheduling

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE over

097 700

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

43

20. ABSTRACT (continued)

of tasks and/or modification of the pipeline; or the collisions can be resolved at the time and place they occur. These alternatives are studied in this work with the objective that the throughput, i.e., average number of tasks processed per unit time, be maximized while providing considerable flexibility in scheduling of tasks.

Collision characteristics of task schedules and pipelines are studied. A methodology for inserting fixed delays in a pipeline to achieve desired collision characteristics is presented.

Collisions can be resolved by letting one task use the segment and storing the rest of the competing tasks in a buffer. Several priority schemes for the buffers have been investigated, analytically for periodic task arrivals and experimentally for random arrivals. The characteristics studied are, queue size, wait time and throughput.

It is shown that the theoretical maximum throughput of a pipeline is attainable with the use of fixed delays or buffers. Moreover it is shown that a substantial degree of freedom in scheduling of tasks is achieved by these methods.

ACCESSION for
White Section [✓]
Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION..........

BY..........
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

UILU-ENG 76-2235

IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS

by

Janek H. Patel

This work was supported in part by the National Science Foundation under Grant MCS 73-03488 A01 and in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

Approved for public release. Distribution unlimited.

D D C
RECEIVED
APR 18 1977
D

ACKNOWLEDGMENTS

I wish to express my gratitude to my advisor Dr. Edward S. Davidson for his guidance, encouragement and genuine concern throughout the course of this research. These years of association with Dr. Davidson will always be remembered and valued.

I also wish to express my appreciation to Mrs. Rose Harris for her immaculate typing of this report.

Finally, I wish to thank all my colleagues at Stanford Electronics Laboratories and Coordinated Science Laboratory for providing an intellectually stimulating atmosphere.

IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS

Janak H. Patel, Ph.D.
Coordinated Science Laboratory

A pipeline as defined here is a collection of segments of hardware which can operate simultaneously. A task flows synchronously from segment to segment for its execution. Each task follows one of several distinct task flow patterns which are assumed to be fixed and known in advance.

It is characteristic of pipelines that a task can be initiated in the pipeline before an earlier initiated task has completed its execution. A problem arises when two or more tasks try to use the same segment at the same time, causing a collision. Such collisions can be avoided by appropriate scheduling of tasks and/or modification of the pipeline; or the collisions can be resolved at the time and place they occur. These alternatives are studied in this work with the objective that the throughput, i.e., average number of tasks processed per unit time, be maximized while providing considerable flexibility in scheduling of tasks.

Collision characteristics of task schedules and pipelines are studied. A methodology for inserting fixed delays in a pipeline to achieve desired collision characteristics is presented.

Collisions can be resolved by letting one task use the segment and storing the rest of the competing tasks in a buffer. Several priority schemes for the buffers have been investigated, analytically for periodic task arrivals and experimentally for random arrivals. The characteristics studied are, queue size, wait time and throughput.

It is shown that the theoretical maximum throughput of a pipeline is attainable with the use of fixed delays or buffers. Moreover it is shown that a substantial degree of freedom in scheduling of tasks is achieved by these methods.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1. Background

Present day computers however powerful they may be, are still inadequate for some problems. Typical of such problems is global weather modeling. Indeed, there always will be problems which will require more and more powerful machines. There are three alternatives (not necessarily mutually exclusive) to obtain higher computational capability. They are, higher speed electronic components, parallel processing and pipeline processing. It is the pipeline or overlap processing which is the subject matter of this research.

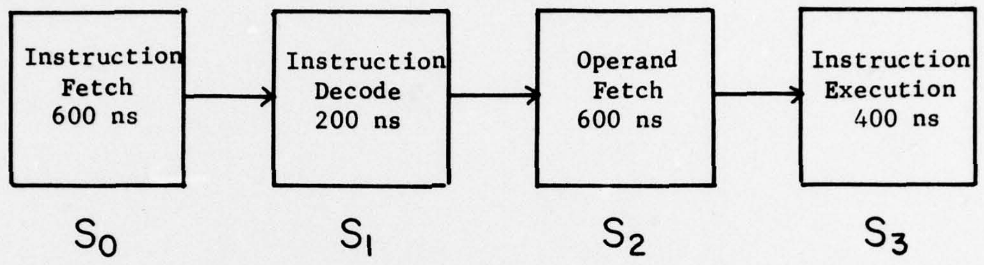
The concept of pipelining is an old one. The bucket brigade used to fight fires of yesteryear was a pipeline. The assembly line used in manufacturing plants is an example of pipelines. Consider an automobile assembly line with n workstations. A partially assembled automobile moves from one workstation to another. Each workstation does a specific job on an auto. Assume that each workstation operates on an auto for t minutes. Then a finished car leaves the line every t minutes even though the total time to assemble a car is t times the number of workstations. The same concept is utilized in computational pipelines.

A pipeline can be defined as a collection of resources called segments which can be kept busy simultaneously. A task once initiated flows from segment to segment for its execution, each segment performing a specific suboperation. The flow of each task is predetermined and fixed. A task once initiated must flow synchronously without preemption or wait, unless internal buffers are provided between segments.

We term a pipeline in which all the tasks have identical flow pattern, a single function pipeline. In a multifunction pipeline there are two or more distinct flow patterns and each task uses one of these flow patterns.

A schematic example of a single function pipeline appears in Figure 1.1.1a. Each segment has an associated output register or latch. The output of the segment is transferred to this register when an appropriate control signal is received. The time shown in each box is the time to perform that particular subtask and includes the time to latch the output. For this illustration, let us ignore any memory conflicts between instruction fetch and operand fetch. The total time to process an instruction in this example is 1800 ns. If no overlap between instructions is provided then the maximum processing rate is one instruction per 1800 ns. However, with overlap permitted it is easy to see that one instruction every 600 ns can be processed.

Thus the effectiveness of pipelining lies in the fact that a task can be initiated before the completion of some previously initiated tasks, resulting in high performance. Moreover, the segments can be special rather than general purpose, resulting in low cost. Thus due to their higher performance per unit cost,pipelines are becoming increasingly common in several recent and proposed computer



(a) Schematic showing the flow of tasks.

SEGMENT \ TIME	0	1	2	3	4	5	6	7	8
S_0	X	X	X						
S_1				X					
S_2					X	X	X		
S_3								X	X

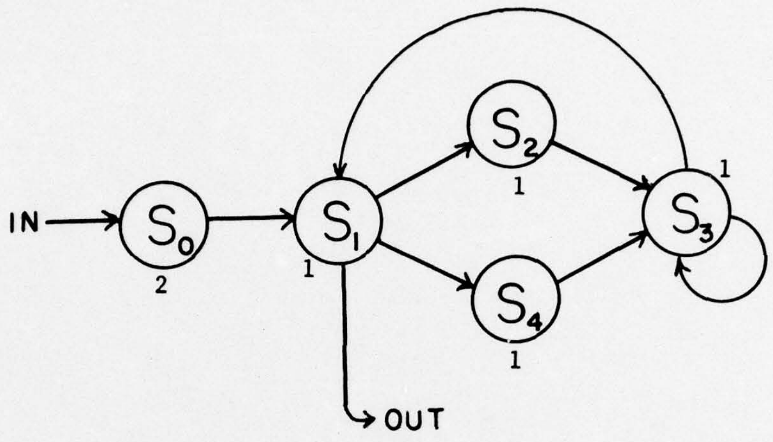
(b) Reservation table.

Figure 1.1.1. Example of a single function pipeline.

systems. Familiar among the pipeline computers are IBM 360/91 [AND67a,AND67b], Texas Instruments-ASC [WAT72,STE73], CDC 7600 [BON69] and CDC STAR-100 [HIN72]. Some others are described in [IBB72,SHA74, PAR74].

It is convenient to define a basic time unit for segment execution times. For example, if the time unit is taken as 200 ns in Figure 1.1.1a then segment times for instruction fetch and operand fetch are each 3 units; for instruction decode, 1 unit; and for instruction execution, 2 units. In general, the time unit is taken to be equal to the greatest common divisor of the execution times of all segments. The system clock is defined in terms of this basic time unit where, one clock cycle equals one time unit. This clock synchronizes the data transfers from segment to segment. This synchronization is required in our model.

The flow in a pipeline can be described more conveniently by a <u>reservation table</u> such as Figure 1.1.1b. Rows of a reservation table correspond to segments and columns correspond to units of time. The function type denoted by a single capital letter is placed in row i and column j (cell (i,j)) of the reservation table if after j units of execution a task of that function type requires segment S_i. We shall use X as a function type in single function pipelines. Figure 1.1.1b is the reservation table of the pipeline of Figure 1.1.1a. The schematic of a slightly more complex pipeline is given in Figure 1.1.2a, its reservation table appears in Figure 1.1.2b. Figure 1.1.3 is an example of a multifunction pipeline with two function types



(a) Schematic of the flow.

	0	1	2	3	4	5	6
S_0	X	X					
S_1			X				X
S_2				X			
S_3					X	X	
S_4				X			

(b) Reservation table.

<u>Figure 1.1.2</u>. A pipeline with parallel computation and feedback.

	0	1	2	3	4
S_0	A		B		B
S_1		AB		B	
S_2	B	A	AB		

Figure 1.1.3. Reservation table of a multifunction pipeline.

A and B. As yet we have not put any restriction on the flow patterns. We may impose some minor restrictions on permissible flow patterns later on. However, no restrictions are assumed on the reusages or sharing of a segment.

Reusage or sharing of a segment is identified in a reservation table by multiple entries of X's or other function types in a single row. It is the reusage or sharing of segments which poses a problem, namely, two or more tasks may attempt to use a segment at the same time causing a collision. For the proper operation of the pipeline either the task initiations must be properly scheduled so that no collisions occur or internal buffers between segments must be provided so that all but one of the tasks are preempted. Previous researchers have concentrated their efforts on finding a schedule for initiations which causes no collisions and gives a high throughput. Before we outline their work some terminology is necessary.

1.2. Pipeline Terminology

A computation step is an indivisible operation by a segment on a task. Thus in Figure 1.1.2 the first computation step is 2 time units long; the next step is one unit long. Two parallel computation steps on segments S_2 and S_4 are each one unit long and two successive steps on S_3 are each one time unit. A knowledge of computation steps, which distinguishes between S_0 and S_3 usage, is relevant to considering possible pipeline redesign.

A usage interval of a segment is defined to be the time interval between two reservations (X's) of that segment. For example in Figure 1.2.1 all usage intervals of S_0 are 2, 3 and 5. Let $\underline{F}$ denote the set of all usage intervals of a pipeline; e.g., $\underline{F} = \{1,2,3,5\}$ for Figure 1.2.1.

An initiation interval of a pair of task initiations is simply the interval in time units between the two time instants of those initiations. A latency is the initiation interval of two successive initiations.

A sequence of task initiations can be completely described by a sequence of latencies. For example, task initiations at time instants 0, 3, 5, 9 and 12 can be described by the latency sequence $\langle 3,2,4,3 \rangle$. Let $\underline{G}$ denote the set of all initiation intervals of an initiation sequence. Thus $\underline{G}$ for latency sequence $\langle 3,2,4,3 \rangle$ is $\{2,3,4,5,6,7,9,12\}$.

Clearly, a collision occurs between two tasks if their initiation interval equals one of the usage intervals of the pipeline [DAV71].

For multifunction pipelines, the usage intervals and initiation intervals are defined similarly. Let $\underline{F}_{XY}$ be the set of usage intervals of all $\langle X,Y \rangle$ pairs in the reservation table. This set can be formed by taking all pairwise distances between an X and a Y which appears to the right of the X in the same row. Here we use X and Y as variables, they take function types as their values; these values need not be distinct. Thus for Figure 1.1.3, the following sets of usage

	0	1	2	3	4	5
S_0	X		X			X
S_1		X	X		X	
S_2			X	X		

Figure 1.2.1. Reservation table of a single function pipeline.

intervals exist:

$$\underline{F}_{AA} = \{1\},\ \underline{F}_{AB} = \{0,1,2,4\},\ \underline{F}_{BA} = \{0,1,2\},\ \underline{F}_{BB} = \{2\}.$$

A multifunction initiation sequence can be described by a latency sequence, where each latency is subscripted by the type of the task being initiated. Thus, for example, initiations of tasks of type A at time instants 0,3,5; of tasks of type B at instants 4,5; and of type C at time instants 7,9 can be described by the latency sequence,

$$\langle \infty_A, 3_A, 1_B, 1_A, 0_B, 2_C, 2_C \rangle.$$

∞_A is just a convenient way of saying that the first initiation is of type A. Two tasks of type A and B are being initiated simultaneously at time instant 5 and therefore the latency between these tasks is 0; in the latency sequence they can be written in either order. A latency of 0 between two tasks of the same type is not permissible.

In a multifunction pipeline a collision occurs if an interval between a task of type X and an earlier initiated task of type Y is equal to the one of the usage intervals of the pair $\langle X,Y \rangle$ in any row of the reservation table. We shall present the formulation of initiation intervals of a multifunction sequence later in Section 2.10.

If a subsequence of latencies repeats itself in an infinite sequence, then the subsequence is termed an <u>initiation cycle</u> or simply a cycle. Thus a cycle $(1_A, 3_B, 2_C, 3_A)$ implies an initiation sequence $\langle \ldots\ 1_A, 3_B, 2_C, 3_A, 1_A, 3_B, 2_C, 3_A, 1_A, 3_B, 2_C, 3_A\ \ldots. \rangle$. In a single function pipeline, a cycle, for example (2,3,2,5), implies a latency sequence,

⟨... 2,3,2,5,2,3,2,5,2,3,2,5,2,3 ....⟩. A cycle with only one latency is called a constant latency cycle, e.g., cycle (4).

An initiation cycle is said to be allowable with respect to a pipeline, if no collisions occur when the cycle is followed for task initiations; conversely we also say that the pipeline is allowable with respect to the cycle. The definition of allowability also applies at finer levels. For example, we say that an initiation interval between two tasks is allowed by a pipeline if no collision occurs; and a usage interval is allowed by a cycle if no collision occurs due to that particular usage interval.

The period p of a cycle is defined as the sum of the latencies of the cycle. The average latency ℓ_a of a cycle is the average of the latencies in a cycle. For example, the period p of cycle (2,3,2,5) is 12 and average latency ℓ_a is 12/4 = 3. Thus cycle (2,3,2,5) implies an average initiation rate of one task every 3 time units. If there is no storage in the pipeline then the output rate equals input rate once the first output occurs. Thus in a pipeline without buffers, the throughput is just the reciprocal of the average latency of the cycle.

Segment utilization is defined in terms of the fraction of the time the segment remains busy. Clearly, the maximum possible utilization of any segment is 100%.

In a single function pipeline if m is the maximum number of X's in any row of the reservation table then there is a segment which remains busy for m time units for every task initiated. Thus no more than one task every m time units can be processed by the pipeline since the average latency of m implies a 100% utilization of some segment. Thus, m is called the lower bound latency of the pipeline.

A single function pipeline is called a straight through pipeline if a task is processed by each segment exactly once. Figure 1.1.1 is an example of such a pipeline. Characteristic of such a pipeline is the absence of any feedback or sharing.

1.3. Review of Related Work

The pipeline form considered here was first proposed by Davidson [DAV71,74,75]. He has analyzed single function pipelines in detail. Subsequently, Shar [SHA72] and Winslow [WIN73] have analyzed exclusively the single function pipelines and Thomas [THO74] has analyzed exclusively multifunction pipelines. Mayeda [MAY75] has suggested an interesting approach using graph theory. They all have one thing in common. They have concentrated their efforts on finding allowable initiation cycles with lowest possible average latency for a given pipeline. Davidson gave a branch-and-bound algorithm for finding such cycles for a given pipeline. As yet no improved algorithms have been found.

A simple control mechanism for collision-free task initiations was devised by Davidson. In a single function pipeline, this control scheme required one shift register. Davidson's shift-register-controller initiates a waiting task as soon as allowed by the pipeline;

that is, as soon as a collision-free flow can be guaranteed through the pipeline. This initiation strategy is called *greedy*. Greedy strategy minimizes the current latency rather than the average latency in the long run. Shar and Winslow have studied the greedy initiation strategy in great detail [SHA72,WIN73]. Any cycle which results from a greedy initiation sequence is called a *greedy cycle*. For a single function pipeline the following inequalities are satisfied.

maximum no. of X's in any row of the reservation table	≤	minimum average latency of any allowable cycle	≤	average latency of any greedy cycle	≤	(size of the usage interval set) + 1

Multifunction pipelines are somewhat complicated. Davidson's shift-register-controller is still applicable for greedy initiations, but the number of shift-registers required is equal to the number of distinct functions. Moreover, the controller must also have a random access to a table of constants. The table has number of bits proportional to the square of the number of functions. However, the controller is still relatively simple and inexpensive, considering the complexity of a multifunction pipeline. Thomas has given an algorithm to find a minimum set of "good" allowable cycles of a multifunction pipeline [THO74]. For any given function mix, a linear combination of good cycles exists which produces the maximum throughput possible with any collision-free strategy.

The principal limitations of the above scheduling strategies are as follows:

1. The assumption that the initiations be allowable is somewhat restrictive. In many situations, there are no allowable cycles which would utilize a pipeline to its fullest extent, i.e. utilize some segment 100%.

2. The requirements for flexibility in scheduling and high throughput are contradictory. For example, greedy control in multi-function pipelines gives sufficient flexibility in initiations. However, some simulation studies have shown that maximum segment utilization can easily be as low as 50%. On the other hand, if an allowable cycle with low average latency is used then any changes in the arrival pattern which do not conform to the cycle cannot be tolerated.

In other related work, Larson [LAR73] describes a methodology for designing straight through pipelines. He gives some guidelines for optimally dividing a functional unit or resource into segments. There also exists some work at the system level and some at the logic level [CHE71,FLY72,HAL72,TOM67].

Other seemingly related work deals with job-shop scheduling. In operations-research terminology our pipeline is a job-shop where segments correspond to machines and tasks to jobs. The job-shop scheduling algorithms, however, are inapplicable to the pipeline model considered here for two reasons. First, job-shops normally assume infinite internal buffers, with few exceptions [RED73], which is impractical for computational pipelines. Second, the complexity of the job-shop scheduling algorithm is combinatorial in the number of tasks and segments, and thus it is practically impossible to schedule more than a few hundred tasks, while the number of tasks (e.g. instructions of a computer) executed in a pipeline might run to some millions per second. We close with a note that even though job-shop scheduling cannot be successfully applied to pipelines, pipeline scheduling can be applied to many practical job-shops, e.g., an assembly line. Pipeline scheduling takes advantage of the pipeline structure and function flow in scheduling and thus has a complexity related primarily to the number of distinct functions rather than to the number of tasks to be scheduled.

1.4. Objectives of this Research

1. To improve the throughput of a pipeline by appropriate scheduling and/or modification of the pipeline.

2. To provide considerable flexibility in scheduling without adversely affecting the throughput of the pipeline.

3. To provide some methodology for designing a cost-effective pipeline.

1.5. An Overview of this Work

In Chapter 2 we present a methodology for inserting fixed delays in a pipeline so that the modified pipeline allows a given cycle. We also present the allowability characteristics of pipelines and cycles. In particular, we present the structure of all possible pipelines allowed by a given cycle.

In Chapter 3 we remove the restriction that the initiations be allowable. Instead we provide internal buffers to resolve collisions. We present the characteristics of various priority schemes under the assumption that the input is periodic. Characteristics considered are the buffer size, throughput and execution delay.

We present some simulation studies of internally buffered pipelines in Chapter 4. The major differences from Chapter 3 are the assumptions that arrivals are random and the buffer sizes are fixed. Moreover, the results of this chapter are experimental rather than analytic. We also present the characteristics of various priority schemes.

Chapter 5 is an attempt at providing a design methodology for cost-effective pipelines. Here we show how the results of Chapters 2, 3 and 4 can be used in the design of a pipeline.

We summarize the main results of this research in Chapter 6. We also present some suggestions for further research in this area.

Chapter 2

PIPELINES WITH DELAYS

2.1. Introduction

The usage intervals of a pipeline determine its allowable initiation sequences. It should thus be possible to design or modify a pipeline to have a specific set of usage intervals which would allow a desired initiation cycle. This is the subject matter of this chapter. We shall investigate the allowability characteristics of pipelines and cycles and use this information to modify a given pipeline to allow a given cycle. The modification is restricted to the addition of noncompute segments which merely provide some fixed delay between selected computation steps.

The actual physical implementation of a noncompute segment need not be considered since it does not alter our treatment. The theory developed is adequate for any physical implementation. However, some comments here on the use of a noncompute segment are useful. Noncompute segments will be used to delay both inputs to computation steps and outputs from computation steps. Sometimes it will be necessary to do both as indicated in the following example.

Suppose in the reservation table of Figure 2.1.1a we want to delay the input to the computation step in row 0, column 2 by two time units and the input to the step in row 2, column 2 by one time unit. The resulting reservation table is shown in Figure 2.1.1b. Each d indicates one unit of delay which we shall refer to as an elemental delay. The

	0	1	2	3	4
S_0	X		X		
S_1		X	X		X
S_2			X	X	

(a)

	0	1	2	3	4	5	6
S_0	X		d	d	X		
S_1		X	X				X
S_2			d	X		X	

(b)

	0	1	2	3	4	5	6
S_0	X		d_1	d_2	X		
S_1		X	X	d_4	d_5		X
S_2			d_3	X	d_6	X	

(c)

<u>Figure 2.1.1</u>. Delaying parallel computation steps in a pipeline.

assignment of elemental delays to physical noncompute segments will be treated later. In the absence of any specific information on precedence we assume that all the steps in one column must be completed before any steps in the next column are initiated. Therefore if the steps of column 2 (of Figure 2.1.1a) are unevenly delayed, then we must store the outputs of some steps so that all the outputs are simultaneously available to the steps of column 3 (of Figure 2.1.1a). These delays are shown in Figure 2.1.1c. The elemental input delays d_1, d_2 and d_3 require d_4, d_5 and d_6 as elemental output delays.

Sections 2.2 through 2.9 deal with single function pipelines. Most of the results concerning single function pipelines can be extended to multifunction pipelines. This extension is made in Section 2.10. In Sections 2.2 and 2.9 we discuss the allowability characteristics of pipelines and cycles. We deal with constant latency cycles in Sections 2.3 through 2.8. Constant latency cycles are treated in considerable detail since the discussion is more straightforward and furthermore motivates the more complicated general case.

2.2. The Allowability of Pipelines and Cycles

In this section we derive a necessary and sufficient condition for the allowability of a pipeline with respect to a given cycle.

For integer x and positive integer y let x mod y denote the remainder when x is divided by y. For some set $\underline{S}$ of integers and positive integer y let us define $\underline{S}$ mod y to be the set formed by replacing each element x of $\underline{S}$ by (x mod y). We may denote $\underline{S}$ mod y as simply $\underline{S}_y$ when the context is clear. The following notation will be used consistently throughout this chapter.

n = the number of initiations in a cycle $(\ell_0,\ell_1,\ldots,\ell_{n-1})$.

p = the period of a cycle $(\ell_0,\ell_1,\ldots,\ell_{n-1}) = \sum_{0\le i<n} \ell_i$.

$\underline{F}$ = the set of usage intervals of a pipeline.

$\underline{G}$ = the set of initiation intervals of a cycle $(\ell_0,\ell_1,\ldots,\ell_{n-1})$.

$= \{g \mid \exists$ integers j,k where $0\le j\le k$ and $g = \sum_{j\le i\le k} \ell_{(i \bmod n)}$.

$\underline{Z}_p$ = the set of integers modulo p.

$\underline{H}_p$ = the complement of the set $\underline{G}$ mod p in $\underline{Z}_p$

$= \underline{Z}_p - \underline{G}_p$.

Note that $\underline{G}$ is an infinite set. However, it is very easy to generate this set from a given cycle, since $\underline{G}$ is trivially related to $\underline{G}_p$.

<u>Example 2.2.1</u>: For cycle (2,3,2,5)

$$n=4 \text{ and } p=12.$$

The given initiation cycle can be written as an infinite latency sequence,

$$\langle 2,3,2,5,2,3,2,5,2,3,2,5,\ldots\ldots\rangle.$$

We can now write the set of initiation intervals of the above sequence as described in Chapter 1.

$\underline{G} = \{2,3,5,7,9,10,12,14,15,17,19,21,22,\ldots\ldots\}$

$\underline{G}_{12} = \{0,2,3,5,7,9,10\}$. □

Note that all elements of $\underline{G}$ greater than p can be generated simply by adding all the positive multiples of p to the elements less than or equal to p. This property is formally described below.

<u>Property 2.2.1</u>: (a) $0 \in \underline{G}_p$ and $qp \in \underline{G}$ $\forall q \geq 1$, always.

(b) if $g \neq 0$ then $g \in \underline{G}_p \Rightarrow g+qp \in \underline{G}$ $\forall q \geq 0$.

<u>Proof</u>: a. The initiation cycle has a period equal to p. Therefore if a task is initiated at time t then another task is initiated at t+p. Therefore the initiation interval $(t+p) - t \in \underline{G}$. Carrying the same argument further, other tasks are initiated at t+2p, t+3p,... Therefore the initiation intervals $p, 2p, 3p, \ldots \in \underline{G}$ or $qp \in \underline{G}$ $\forall q \geq 1$. It immediately follows that $(qp \bmod p) \in \underline{G}_p$; that is, $0 \in \underline{G}_p$.

b. If $g \in \underline{G}_p$ then $\exists$ some integer $r \geq 0$ such that $g+rp \in \underline{G}$. In other words, there exist two tasks which are initiated g+rp units apart, say at time t and t+g+rp. Then again by the same argument as in (a) above, tasks are also initiated at t+p, t+2p,... t+rp,... and t+g+rp, t+g+(r+1)p, t+g+(r+2)p,.... The tasks initiated at time t+rp and t+g+rp are really two different tasks because $g \neq 0$. Therefore the intervals $(t+g+rp)-(t+rp), (t+g+(r+1)p)-(t+rp), \ldots \in \underline{G}$. Thus $g+qp \in \underline{G}$ $\forall q \geq 0$. Q.E.D.

<u>Property 2.2.2</u>: If $g \neq 0$ then

$$g \in \underline{G}_p \iff (p-g) \in \underline{G}_p.$$

<u>Proof</u>: By Property 2.2.1b $g \in \underline{G}_p \Rightarrow g \in \underline{G}$. Then there are two tasks which are initiated g time units apart. Suppose the two tasks are initiated at time t and t+g. By the periodicity of the initiations,

there is also a task which is initiated at time t+p. Thus the interval $(t+p)-(t+g) \in \underline{G}$. Therefore

$$g \in \underline{G}_p \Rightarrow (p-g) \in \underline{G}$$
$$\Rightarrow (p-g) \bmod p \in \underline{G}_p.$$

Since $1 \leq g \leq (p-1)$, $(p-g) \bmod p = p-g$ and thus $(p-g) \in \underline{G}_p$. The reverse implication is also true, since $(p-g) \neq 0$ and $p-(p-g) = g$. Therefore $g \in \underline{G}_p \iff (p-g) \in \underline{G}_p$. Q.E.D.

The same property as above also holds for the set $\underline{H}_p$, the complement of $\underline{G}_p$ in $\underline{Z}_p$, as shown below.

<u>Property 2.2.3</u>: $h \in \underline{H}_p \iff (p-h) \in \underline{H}_p$.

<u>Proof</u>: First note that $h \neq 0$ since by Property 2.2.1a $0 \in \underline{G}_p$ always, thus $1 \leq h \leq (p-1)$. Suppose $h \in \underline{H}_p$ but $(p-h) \notin \underline{H}_p$, then $(p-h) \in \underline{G}_p$. By Property 2.2.2, this implies $h \in \underline{G}_p$ a contradiction since, $h \in \underline{H}_p$. Therefore

$$h \in \underline{H}_p \Rightarrow (p-h) \in \underline{H}_p.$$

And since $p-(p-h) = h$, we have

$$h \in \underline{H}_p \iff (p-h) \in \underline{H}_p.$$ Q.E.D.

The following theorem establishes a necessary and sufficient condition for allowability.

<u>Theorem 2.2.1</u>: A cycle with period p and initiation interval set $\underline{G}$ is allowed by a pipeline with usage interval set $\underline{F}$, if and only if

$$\underline{F}_p \cap \underline{G}_p = \underline{\Phi}.$$

<u>Proof</u>: A cycle is allowable if and only if no collision occurs when the cycle is followed. A collision between two tasks occurs if and only if their initiation interval is the same as one of the usage intervals of the pipeline. Thus the cycle is allowable if and only if

$\underline{F} \cap \underline{G} = \underline{\Phi}$, where $\underline{\Phi}$ is the empty set.

To complete the proof it is sufficient to show that

$$\underline{F} \cap \underline{G} = \underline{\Phi} \iff \underline{F}_p \cap \underline{G}_p = \underline{\Phi}$$

or, equivalently

$$\underline{F} \cap \underline{G} \neq \underline{\Phi} \iff \underline{F}_p \cap \underline{G}_p \neq \underline{\Phi}.$$

(i) Suppose $\underline{F} \cap \underline{G} \neq \underline{\Phi}$ then for some $x \in (\underline{F} \cap \underline{G})$

$$x \in \underline{F} \Rightarrow (x \bmod p) \in \underline{F}_p$$
$$x \in \underline{G} \Rightarrow (x \bmod p) \in \underline{G}_p$$

and therefore $\underline{F}_p \cap \underline{G}_p \neq \underline{\Phi}$.

(ii) To prove the backward implication,

suppose $\underline{F}_p \cap \underline{G}_p \neq \underline{\Phi}$ then for some $x \in \underline{F}_p \cap \underline{G}_p$,

if $x \neq 0$ then,

$x \in \underline{F}_p \Rightarrow x + jp \in \underline{F}$ for some integer $j \geq 0$

$x \in \underline{G}_p \Rightarrow x + ip \in \underline{G}$ for all integers $i \geq 0$ (by Property 2.2.1).

Therefore

$$x + jp \in \underline{F} \cap \underline{G}.$$

Similarly, if $x = 0$ then

$x \in \underline{F}_p \Rightarrow x + jp \in \underline{F}$ for some integer $j \geq 1$.

$x \in \underline{G}_p \Rightarrow x + ip \in \underline{G}$ for all integers $i \geq 1$ (by Property 2.2.1).

Therefore

$$x + jp \in \underline{F} \cap \underline{G}.$$

Thus in either case

$$\underline{F} \cap \underline{G} \neq \underline{\Phi}.$$

Therefore

$$\underline{F} \cap \underline{G} \neq \underline{\Phi} \iff \underline{F}_p \cap \underline{G}_p \neq \underline{\Phi}. \qquad \text{Q.E.D.}$$

An equivalent to the following corollary appeared in [SHA72].

Corollary 2.2.1.1: A constant latency cycle (ℓ) is allowed by a pipeline if and only if no usage interval of the pipeline is an integral multiple of ℓ.

Proof: Immediate, because period $p = \ell$ and $\underline{G}_p = \{0\}$. Cycle (ℓ) is allowable if and only if $(\underline{F} \bmod \ell) \cap \{0\} = \underline{\Phi}$. Q.E.D.

From the above theorem we may attach the following meaning to the set $\underline{H}_p$, the complement set of $\underline{G}_p$. A cycle is allowable if and only if $\underline{F}_p \cap \underline{G}_p = \underline{\Phi}$, or equivalently if and only if $\underline{F}_p \subseteq \underline{H}_p$. Hence $\underline{H}_p$ is the set of allowable usage intervals modulo p with respect to the given cycle.

2.3. Constant Latency Cycles

Sections 2.3 through 2.8 deal exclusively with constant latency cycles. Given a constant latency cycle, it is possible to modify a pipeline by replicating some segments such that the pipeline becomes allowable. For example one can always replicate the segments a sufficient number of times such that no segment is used more than one time unit per task, making the pipeline allowable for any cycle. But it is not apparent when one can make a pipeline allowable simply by introducing some fixed delays. We address this problem in the following theorem. A theorem similar to this was proven by Shar [SHA72]. We shall see later in Section 2.9 that there exists a somewhat similar but more restrictive statement concerning arbitrary cycles.

<u>Theorem 2.3.1</u>: For a given constant latency cycle (ℓ), any pipeline can be reconfigured using noncompute segments such that the pipeline is allowable, if and only if the lower bound m of the pipeline is less than or equal to ℓ.

<u>Proof</u>: (i) First assume $m \leq \ell$. By Corollary 2.2.1.1, a constant latency cycle (ℓ) is allowable if no usage interval (i.e., distance between any two X's in any row of the reservation table of the pipeline) is an integral multiple of ℓ. For a particular row the distance between an X in the i^{th} column and an X in the j^{th} column is $|j-i|$. This distance is a multiple of ℓ if and only if $i \equiv j \pmod{\ell}$.

Let us relabel each column by its modulo ℓ representative, that is, we relabel a column marked i with ($i \bmod \ell$). Now if we make certain that no two X's in a row have the same column-label then we have ensured that no usage interval is an integral multiple of ℓ. We give the following construction to show that this can always be done by delaying some computation steps.

Scan the columns of the relabeled reservation table from left to right. Then scan each column from top to bottom. As each X is found, it may be moved to the right to ensure that its column-label is different from the column-labels of all previously found X's in that row. This can always be done since there are ℓ distinct labels, but there are no more than m X's in a row and $m \leq \ell$. Moving an X to the right by k columns is equivalent to delaying that computation step by k time units. (If any X's in the column being processed are moved then the unscanned columns of the reservation table are moved to the right to preserve the order of computation steps as implied by the original table.)

(ii) The reverse implication is straightforward. From the definition of the lower bound m, no pipeline can ever be allowable by insertion of delays if $m > \ell$. Q.E.D.

Figure 2.3.1 illustrates the use of the construction described in the proof of the above theorem. Figure 2.3.1a is the original reservation table and Figure 2.3.1b is the reconfigured table for constant latency $\ell = 4$. Figure 2.3.1c shows the positions of elemental delays marked as "d." The connectors between d and x indicate whether the input or the output of that step is delayed. For consistency, we shall write the elemental delay at the input side of the computation step except when it is necessary to use delay at the output side also, as was described in Section 2.1.

If one noncompute segment is used for each elemental delay then the reservation table in Figure 2.3.1d results, where D_1 through

	0	1	2	3	4	5	6	7	8	9
S_0	X				X			X		X
S_1		X				X	X		X	
S_2			X	X				X	X	

(a)

	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
S_0	X					X					X					X
S_1		X					X	X					X			
S_2			X	X					X					X		

(b)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_0	X				d_1	X			d_2	d_3	X				d_4	X
S_1		X					X	X				d_5	X	d_6		
S_2			X	X					X	d_7	d_8	d_9	d_{10}	X		

(c)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_0	X					X					X					X
S_1		X					X	X					X			
S_2			X	X					X					X		
D_1					X											
D_2									X							
D_3										X						
D_4															X	
D_5												X				
D_6														X		
D_7										X						
D_8											X					
D_9												X				
D_{10}													X			

(d)

Figure 2.3.1. Making a pipeline allowable with respect to cycle (4).

D_{10} are noncompute segments. However, some of the elemental delays can be provided by a single physical noncompute segment. Thus resource sharing is discussed in the following section.

2.4. Sharing of Noncompute Segments

First we shall consider the sharing with some restriction. We assume that a noncompute segment cannot be shared by more than one elemental delay during a single time unit and that an elemental delay may not use more than one noncompute segment.

Note that datapath width may vary from one computation step to another and therefore the datapath widths of elemental delays also may vary. However, this does not prevent several elemental delays from sharing a single noncompute segment, as long as the datapath width of the noncompute segment is greater than or equal to that of each of the elemental delays. In order to separate the issues involved, our only concern now is how sharing can be done without affecting the allowability of the pipeline for the given cycle. Path width and "bit-segments" are considered later in this section.

Let $\langle d_1, d_2, \ldots, d_k \rangle$ be the set of elemental delays ordered arbitrarily in some reconfigured reservation table, and let $\langle t_1, t_2, \ldots, t_k \rangle$ be their corresponding set of column numbers. That is, d_i is located in the column labeled t_i of the reconfigured reservation table, where the columns are indexed 0,1,2,... from left to right in increasing order.

For example the reconfigured table in Figure 2.3.1c has $\langle d_1, d_2, d_3, \ldots, d_{10} \rangle$ as the set of elemental delays and $\langle 4,8,9,14,11,13,9,10,11,12 \rangle$ as the corresponding ordered set of column numbers.

A single noncompute segment can be shared by elemental delays d_i and d_j without affecting the allowability of a constant latency cycle (ℓ), if and only if d_i and d_j are not an integral multiple of ℓ apart (by Corollary 2.2.1.1). That is $|t_i - t_j| \bmod \ell \neq 0$. If $t_i \equiv t_j \pmod{\ell}$ then we call d_i and d_j incompatible because they cannot share the same noncompute segment. The relation "incompatible with" defined by "d_i incompatible with d_j if $t_i \equiv t_j \pmod{\ell}$" is an equivalence relation on the set $\langle d_1, d_2, \ldots, d_k \rangle$. This relation partitions the set of delay elements into equivalence classes. We call these classes incompatibility classes.

No two members of an incompatibility class can share a single noncompute segment. Any two or more elemental delays, each belonging to a distinct incompatibility class, can always share the same segment.

It follows immediately that the size of the largest incompatibility class is the minimum number of noncompute segments required to provide all the elemental delays. Also note that ℓ or fewer elemental delays appearing in successive columns can always share a single noncompute segment, because the greatest separation between any two such elemental delays is less than ℓ and therefore no usage interval can be a multiple of ℓ.

Take again the example of Figure 2.3.1c, which was reconfigured for constant latency 4. The incompatibility classes of the set of elemental delays are

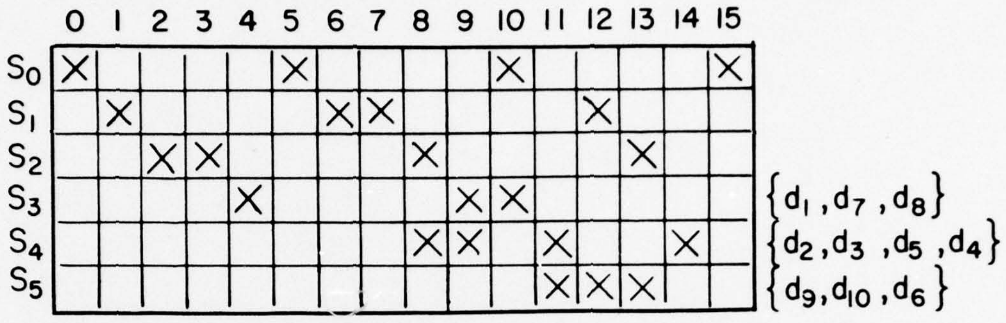
$$\{d_1,d_2,d_{10}\},\{d_3,d_6,d_7\},\{d_4,d_8\},\{d_5,d_9\}.$$

The size of the largest class is 3, therefore 3 noncompute segments are required to provide elemental delays d_1 thru d_{10}. Of many possible sharing configurations one is shown in Figure 2.4.1 where S_3, S_4 and S_5 are noncompute segments.

In some situations the type of segment sharing proposed above may not be suitable. One can for example easily formulate a "bit sharing" scheme. We next present such a scheme, which can easily be substituted for segment sharing. The discussion in later sections, however, reverts to segment sharing since segment sharing illustrates our fundamental points and is notationally far simpler.

Define a noncompute bit-segment to be a one bit wide noncompute segment, similarly an elemental bit-delay is a one bit wide elemental delay. Thus an elemental delay d_i which is w_i bits wide can be thought of as being made up of w_i elemental bit-delays. The problem now reduces to that of sharing of noncompute bit-segments by elemental bit-delays. This problem is similar to the previous problem.

If the elemental delays d_i and d_j are incompatible then so are the elemental bit-delays of d_i and d_j. The elemental bit-delays of d_i may be denoted by $d_{i1},d_{i2},\ldots,d_{iw_i}$. The incompatibility classes of elemental bit-delays may be formed directly from the incompatibility



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
S_0	X					X					X					X	
S_1		X					X	X					X				
S_2			X	X					X					X			
S_3					X					X	X						$\{d_1, d_7, d_8\}$
S_4									X	X		X			X		$\{d_2, d_3, d_5, d_4\}$
S_5												X	X	X			$\{d_9, d_{10}, d_6\}$

<u>Figure 2.4.1</u>. Assignment of elemental delays to noncompute segments.

classes of elemental delays already formed earlier. Thus for example, the class $\{d_3, d_6, d_7\}$ formed earlier in the discussion of Figure 2.3.1d gives,

$$\{d_{31}, d_{32}, \ldots, d_{3w_3}, d_{61}, d_{62}, \ldots, d_{6w_6}, d_{71}, d_{72}, \ldots, d_{7w_7}\}.$$

In this manner we can form all the bit incompatibility classes. These classes have the same properties as those of incompatibility classes of elemental delays, namely, no two elemental bit-delays from the same class can share a noncompute bit-segment and two or more bit-delays each contained in a distinct incompatibility class can always share a single noncompute bit-segment. Thus the size of the largest bit incompatibility class is the minimum number of noncompute bit-segments required to provide all the bit-delays.

2.5. Defining the Optimum Solution

The construction described in the proof of Theorem 2.3.1 is merely an existence proof, it does not necessarily provide a 'good' solution. This construction when used on the reservation table of Figure 2.3.1a produced the table of Figure 2.3.1c. This solution has 10 elemental delays. If the criterion for goodness is the fewest elemental delays then the reservation table of Figure 2.5.1 with only two elemental delays is a better solution to the same problem. Thus there may be many solutions to the same problem. In this section we define an optimum solution. In the next two sections we discuss how to

	0	1	2	3	4	5	6	7	8	9	10	11
S_0	X					d_1	X			X		X
S_1		X						X	X		X	
S_2			d_2	X	X					X	X	
	0	1	2	3	0	1	2	3	0	1	2	3

<u>Figure 2.5.1</u>. An optimum solution to the problem of making the pipeline of Figure 2.3.1a allowable with respect to cycle (4).

obtain such optimum solutions. Two important parameters in such a solution are added execution time and cost.

The increase in execution time is caused by the delaying of some computation steps. For example the solution of Figure 2.3.1c takes 6 time units more than the original execution time of 10 units. If the pipeline can be kept full by initiating tasks at the maximum rate, then the throughput is the same as the input rate irrespective of the added execution delay. However, in some situations it becomes necessary to empty the pipeline before a new task can be initiated. For example, consider some tasks which are logically dependent; that is, some task can be initiated only after the execution of some previously initiated task is completed. Another example of such dependency is conditional branching; that is, the outcome of some operation determines which one of two or more streams of tasks should be initiated. Thus the total execution time can become an important parameter in determining the overall throughput. In such situations we would like to minimize the added execution delay when reconfiguring a reservation table by the use of added noncompute segments.

The second parameter is the cost, that is, the cost of adding noncompute segments. Some of the factors which contribute significantly to this cost are: the cost of the noncompute segments themselves, the cost of additional control and the cost of additional data-paths. The number of noncompute segments may be reduced by utilizing the sharing techniques described in the previous section. However, sharing usually adds more datapaths. Also it makes data routing among segments more complex. On the other hand some types of sharing are simple and profitable. For example, as mentioned before, ℓ or fewer successive elemental delays can always share a single noncompute segment. Note that such sharing is profitable since it can be implemented by a single register with an appropriate clock without creating any additional datapaths. Furthermore, an input elemental delay may be replaced by an output delay of the preceding computation step and vice versa. Though these configurations are logically equivalent, they are not necessarily equivalent in cost. It is clear that to compute the cost difference between any two configurations of a pipeline would require extensive description of all datapaths and their widths, the actual techniques used for data routing (or communication among segments) and the actual implementation of noncompute segments together with the formulas for computing the cost from these parameters. A formal cost model can be formed for a specific case, but it would be very cumbersome and of doubtful accuracy in the general case.

From now onwards we shall concern ourselves only with minimizing execution time, and an optimum solution will be defined to be a solution having minimum execution delay.

The algorithm to be presented in Section 2.7, is capable of producing all the solutions having minimum execution delay. Also it is possible to produce solutions having execution delay below a given bound. A designer can then choose a solution out of these to optimize secondary objectives, such as cost.

2.6. Formulation of the Optimization Problem

The problem of making a pipeline allowable for some constant cycle (ℓ) in an optimal fashion can be formally stated as:

Minimize added execution delay subject to

$$\{\underline{F} \bmod p\} \cap \{0\} = \underline{\Phi}$$

where, $\underline{F}$ is the set of usage intervals of the reconfigured reservation table and p is the period of the cycle and is equal to ℓ. This constraint comes directly from Corollary 2.2.1.1.

The following two examples show how we can formulate the problem for a given reservation table. First, some definitions.

Let D be the added execution delay and let d_{ij} and d'_{ij} be the number of elemental delays to be inserted respectively at the input and the output of an X originally in row i and column j (referred to as cell (i,j)) of the reservation table. If no X occurs in cell (i,j) of the table then d_{ij} and d'_{ij} are defined to be zero.

Example 2.6.1: For the reservation table of Figure 2.6.1 we can express the added execution delay, D and the constraints as follows. We need not include those variables d_{ij} and d'_{ij} which are zero by definition.

$$D = d_{00} + d_{11} + d_{22} + d_{23} + d_{14} + d_{05} + d_{06} + d_{17}$$

and the constraints are,

integer $d_{ij} \geq 0$ and

1. $(d_{11}+d_{22}+d_{23}+d_{14}+d_{05}+5) \bmod p \neq 0$ $\langle X_{00},X_{05}\rangle$
2. $(d_{06}+1) \bmod p \neq 0$ $\langle X_{05},X_{06}\rangle$
3. $(d_{11}+d_{22}+d_{23}+d_{14}+d_{05}+d_{06}+6) \bmod p \neq 0$ $\langle X_{00},X_{06}\rangle$
4. $(d_{22}+d_{23}+d_{14}+3) \bmod p \neq 0$ $\langle X_{11},X_{14}\rangle$
5. $(d_{05}+d_{06}+d_{17}+3) \bmod p \neq 0$ $\langle X_{14},X_{17}\rangle$
6. $(d_{22}+d_{23}+d_{14}+d_{05}+d_{06}+d_{17}+6) \bmod p \neq 0$ $\langle X_{11},X_{17}\rangle$
7. $(d_{23}+1) \bmod p \neq 0$ $\langle X_{22},X_{23}\rangle$

The pair of X's in the triangular brackets indicate that the constraint on the left resulted from that particular pair. □

Notice that we did not use the variable d'_{ij} in any of the above constraints. This is because the reservation table did not have more than one X in any column and therefore the output of a step is identical to the input of the next step and output delays are not required. The next example is more general and we will need to insert delays at the output of some computation steps. Output delays were discussed in Section 2.1.

<u>Example 2.6.2</u>: For the reservation table of Figure 2.6.2 we can express the added execution delay D and the constraints as follows. Again we shall omit those variables d_{ij} and d'_{ij} which are zero by definition.

$$D = \max\{d_{00},d_{10},d_{20}\} + d_{01} + d_{22} + \max\{d_{13},d_{23}\} + \max\{d_{04},d_{14}\}.$$

The constraints are,

$$\text{integer } d_{ij} \geq 0 \quad \text{and}$$

	0	1	2	3	4	5	6	7
S_0	X					X	X	
S_1		X			X			X
S_2			X	X				

Figure 2.6.1. Reservation table for Example 2.6.1.

	0	1	2	3	4
S_0	X	X			X
S_1	X			X	X
S_2	X		X	X	

Figure 2.6.2. Reservation table for Example 2.6.2.

1. $(d'_{00} + d_{01} + 1) \bmod p \neq 0$ $\langle X_{00}, X_{01} \rangle$
2. $(d_{22} + \max\{d_{13}, d_{23}\} + d_{04} + 3) \bmod p \neq 0$ $\langle X_{01}, X_{04} \rangle$
3. $(d'_{00} + d_{01} + d_{22} + \max\{d_{13}, d_{23}\} + d_{04} + 4) \bmod p \neq 0$ $\langle X_{00}, X_{04} \rangle$
4. $(d'_{10} + d_{01} + d_{22} + d_{13} + 3) \bmod p \neq 0$ $\langle X_{10}, X_{13} \rangle$
5. $(d'_{13} + d_{14} + 1) \bmod p \neq 0$ $\langle X_{13}, X_{14} \rangle$
6. $(d'_{10} + d_{01} + d_{22} + \max\{d_{13}, d_{23}\} + d_{14} + 4) \bmod p \neq 0$ $\langle X_{10}, X_{14} \rangle$
7. $(d'_{20} + d_{01} + d_{22} + 2) \bmod p \neq 0$ $\langle X_{20}, X_{22} \rangle$
8. $(d_{23} + 1) \bmod p \neq 0$ $\langle X_{22}, X_{23} \rangle$
9. $(d'_{20} + d_{01} + d_{22} + d_{23} + 3) \bmod p \neq 0$ $\langle X_{20}, X_{23} \rangle$

where

$$d'_{00} = \max(d_{00}, d_{10}, d_{20}) - d_{00}$$
$$d'_{10} = \max(d_{00}, d_{10}, d_{20}) - d_{10}$$
$$d'_{20} = \max(d_{00}, d_{10}, d_{20}) - d_{20}$$
$$d'_{13} = \max(d_{13}, d_{23}) - d_{13}. \qquad \square$$

In general we can express the added execution delay D and the constraints in the following manner. Let I be the number of rows and J be the number of columns in the given reservation table, then

$$D = \sum_{0 \leq j < J} \max_{0 \leq i < I} (d_{ij}) \tag{2.6.1}$$

and the constraints are

$$\text{integer } d_{ij} \geq 0,$$

$$(d'_{ab} + \sum_{b < j < c} \max_{0 \leq i < I} (d_{ij}) + d_{ac} + (c-b)) \bmod p \neq 0$$
$$\text{for each pair } \langle X_{ab}, X_{ac} \rangle \text{ with } c > b. \tag{2.6.2}$$

where

$$d'_{ab} = \max_{0 \leq i < I} (d_{ib}) - d_{ab}. \qquad (2.6.3)$$

If there is no X in the cell (i,j) of the reservation table then the variable d_{ij} should be assigned the value zero in all of the above expressions.

In the constraints the term d'_{ab} is the number of elemental delays introduced at the output of computation step X_{ab}. The variable d'_{ab} need not appear explicitly in the constraints because it is expressible in terms of the input delays. The variable d_{ac} is the number of input elemental delays at step X_{ac}. The input delay plus the output delay for column j is given by $\max_i d_{ij}$. Therefore, the sum term in each constraint is the effect of inserted delays in the intervening columns between X_{ab} and X_{ac}. Finally the term (c-b) is the separation which existed between X_{ab} and X_{ac} before the insertion of any delays. The objective function is simply the added column delay summed over all columns.

While forming the constraints, those variables must be set to zero where it is not possible to introduce a delay. Take for example the table in Figure 2.6.1. In the row of segment S_2 there are two successive X's. These two X's might have resulted from a single indivisible computation step which is 2 units long. In such a case variable d_{23} would have to be set to zero to preclude possible preemption of an indivisible computation step. Setting variables to zero in this manner does not destroy the existence of a solution. This should become obvious when we discuss allowable rows of a cycle in Section 2.9.

2.7. Obtaining the Optimum Solution

There is no analytic solution to the optimization problem presented in the previous section, except for a special but important class of pipelines presented in the next section. The solution space is finite because each d_{ij} need only take integer values between 0 and p-1, as all the constraints are given in modulo p arithmetic. This places an upper bound on the added execution time equal to $(p-1)\cdot J$, where J is the number of columns in the reservation table. Moreover the objective function D is nondecreasing in d_{ij}; that is, any addition of an elemental delay either increases the execution delay or leaves it unchanged. These properties allow a branch-and-bound search technique. We present below such a branch-and-bound algorithm.

Algorithm 2.7.1: Let the number of X's in the reservation table be x and let the x variables, d_{ij}, be stored in any arbitrary order in a one dimensional array V. Let D(i) represent the value of the objective function for given values of V(1) through V(i), with V(i+1) through V(x) taken to be 0.

```
B1.  [Initialize] $i \leftarrow 0$; BOUND $\leftarrow (p-1)\cdot J$;
B2.  [Advance] $i \leftarrow i+1$; $V(i) \leftarrow 0$;
B3.  [Check bounds and constraints]  if $(V(i) = p)$ or $(D(i) > \text{BOUND})$ then
         go to B6; if a completely assigned constraint is violated then go
```

to B5;

B4. [Solution found?] if $i < x$ then go to B2 else output the solution $V(1)$ through $V(x)$ and $D(x)$; BOUND $\leftarrow D(x)$;

B5. [Try another value] $V(i) \leftarrow V(i)+1$; go to B3;

B6. [Backtrack] $i \leftarrow i-1$; if $i > 0$ then go to B5 else terminate the algorithm. □

The last value of BOUND is the minimum value of the objective function over all possible solutions and therefore the output solutions meeting this bound are all the minimum added delay solutions. Explanation of some of the steps in the above is as follows.

In step B3, if $V(i) = p$ then we have assigned all possible values to $V(i)$. If the current value of the objective function exceeds BOUND then we need not proceed since the objective function can only increase further. In either case we must backtrack. If a completely assigned constraint is violated then the current value of $V(i)$ is not acceptable and therefore we must try another value. Note that the constraints which consist only of variables $V(1)$ through the current variable $V(i)$, are completely assigned. Out of these we need to check only those constraints which actually have $V(i)$ as a variable, since the other constraints would have been verified previously.

In step B4, if $i < x$ then there are still more variables to be assigned some values. If $i = x$ then all variables have been assigned successfully without violating a constraint or bound and therefore we have a solution.

In step B6, while backtracking, if no variables are left then all possible solutions have been either fully developed or rejected for exceeding BOUND or violating constraints. We thus terminate the algorithm.

The above algorithm generates all the minimum delay solutions. If only one optimum solution is desired then the condition $D(i) > \text{BOUND}$ in step B3 should be changed to $D(i) \geq \text{BOUND}$.

Another possibility is that we start the algorithm with some specific value for BOUND and do not update it later, that is, remove the statement $\text{BOUND} \leftarrow D(x)$ in step B4. Then the algorithm generates all solutions which have their objective function value less than or equal to the fixed value of BOUND. This version would be particularly useful if a designer has some secondary objectives for his solutions.

A complete example with the constraints and the backtrack tree is given in Figure 2.7.1. The variables, d_{ij}, have been retained in the figure for simplicity. B is the variable BOUND, '$>B$' indicates that the bound has been exceeded, and '$\not a$' indicates that the constraint (a) has been violated.

Algorithm 2.7.1 is remarkably efficient in our limited experience. For example for one 20 variable problem with a potential 10^{14} nodes in the tree, only 10^{4} nodes were expanded and an optimum solution was obtained in 40 seconds on an IBM 360/67. For a particular class of problems, the technique of Knuth [KNU75] may be applicable to estimate the complexity of the algorithm.

	0	1	2
S_0	X		X
S_1	X	X	

Make the pipeline allowable for cycle (2):

$$\underline{H} \bmod 2 = \{1\}$$

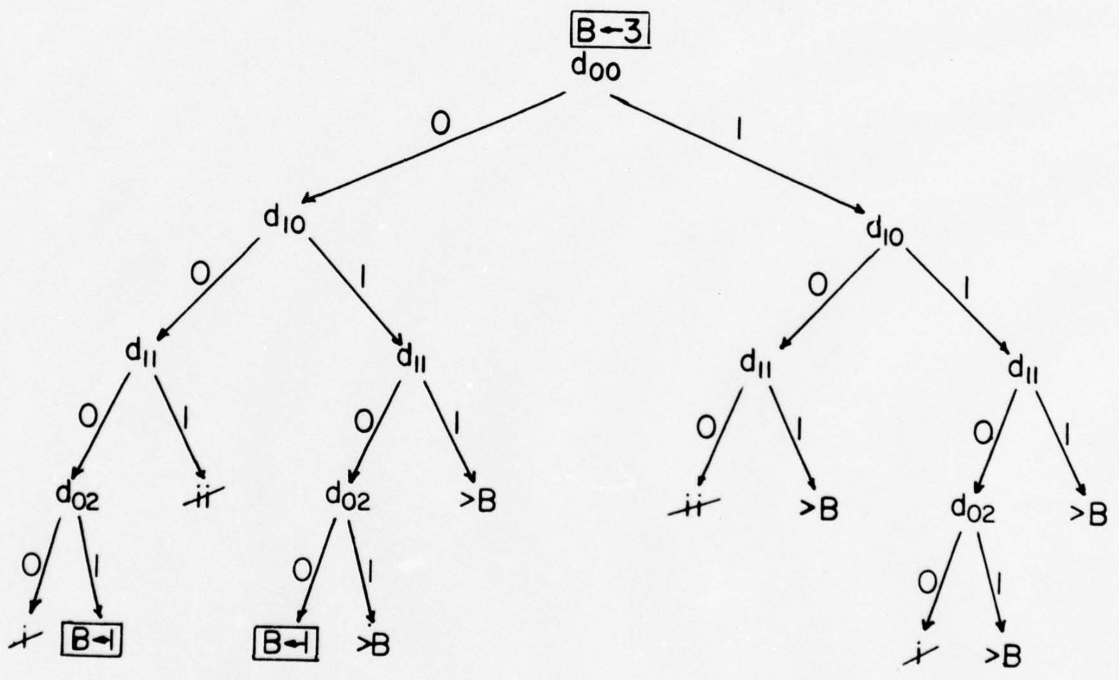
Added delay:

$$D = \max\{d_{00}, d_{10}\} + d_{11} + d_{02}$$

Constraints:

(i) $[2 + \max\{d_{00}, d_{10}\} - d_{00} + d_{11} + d_{02}] \bmod 2 \in \{1\}$.

(ii) $[1 + \max\{d_{00}, d_{10}\} - d_{10} + d_{11}] \bmod 2 \in \{1\}$.



Optimum solutions are: 1. $d_{00} = d_{10} = d_{11} = 0$, $d_{02} = 1$.

2. $d_{00} = d_{11} = d_{02} = 0$, $d_{10} = 1$.

Figure 2.7.1. A branch-and-bound search for optimum solutions.

2.8. Looping Pipelines

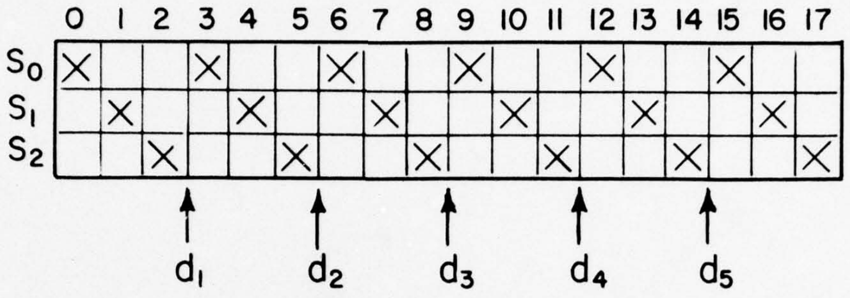
A looping pipeline is characterized by a reservation table which reserves the segments for one time unit each in some sequence and then repeats this sequence several times. This special class of pipelines is of considerable importance as is indicated by their common occurrence; e.g., in the floating point multiply/divide units of the IBM 360/91 and the Texas Instruments ASC. Davidson [DAV71] and Shar [SHA74] have shown that a greedy initiation strategy for this class of pipelines always achieves maximum throughput but possibly only with a non-constant latency cycle. Here we are interested in using constant latency cycles.

The reservation table of Figure 2.8.1a is an example of a looping pipeline. This pipeline has a lower bound latency = 6. The cycle (1,1,16) with average latency 6 is allowable for this pipeline but the constant latency cycle (6) is not.

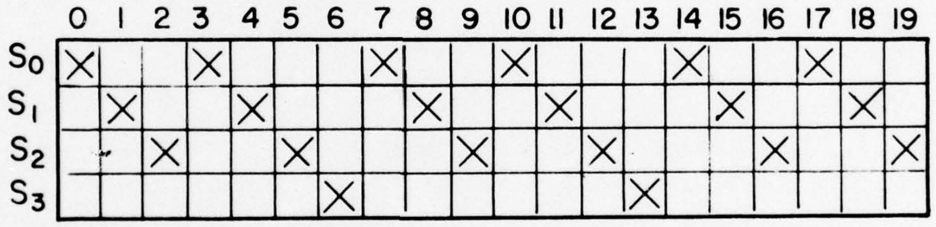
All the segments in a looping pipeline have identical usage patterns and hence each segment has the same set of usage intervals. This property simplifies the problem considerably, as the following results indicate.

Lemma 2.8.1.1: While making a looping pipeline allowable with respect to a given cycle, by adding the least possible execution delay, it is sufficient to consider only those delays which are placed between successive iterations (such as $d_1, \ldots d_5$ in Figure 2.8.1a).

Proof: All the rows of a reservation table of a looping pipeline have identical usage patterns. Therefore an optimum way of inserting



(a)



(b)

<u>Figure 2.8.1</u>. Making a looping pipeline allowable with respect to a constant latency cycle.

delays in one row is also optimum for all other rows. Pick an arbitrary row and find a solution which adds fewest possible delays to this row and makes it allowable. This solution can be applied to the whole reservation table such that the delays fall between the iterations. This insures that the sets of usage intervals of all rows are changed identically. Thus all rows are made allowable optimally by using only those delays which are placed between successive iterations. Q.E.D.

<u>Theorem 2.8.1</u>: For a looping pipeline with lower bound average latency m and number of segments s, there exists the following assignment of values to delays $d_1, d_2, \ldots d_{m-1}$ (see Figure 2.8.1a) so that the cycle (ℓ), for $\ell \geq m$, is allowable and the added execution delay is a minimum:

$$d_q = d_{2q} = d_{3q} = \cdots\cdots = d_{\left\lceil\frac{m-q}{q}\right\rceil q} = 1 \quad \text{all other } d_i = 0.$$

Where q is the smallest positive integer such that ℓ divides qs; that is, $q = \frac{\ell}{\gcd(\ell,s)} = \frac{\mathrm{lcm}(\ell,s)}{s}$ where gcd stands for greatest common divisor and lcm for least common multiple.

<u>Proof</u>: We need to satisfy the following constraints in order to make the pipeline allowable for cycle (ℓ).

$$\begin{array}{l}
(d_1 + s) \bmod \ell \neq 0 \\
(d_2 + s) \bmod \ell \neq 0 \\
\vdots \\
(d_{m-1} + s) \bmod \ell \neq 0 \\
\hline
(d_1 + d_2 + 2s) \bmod \ell \neq 0 \\
(d_2 + d_3 + 2s) \bmod \ell \neq 0 \\
\vdots \\
(d_{m-2} + d_{m-1} + 2s) \bmod \ell \neq 0 \\
\hline
\vdots \qquad \vdots \\
\hline
(d_1 + d_2 + \cdots + d_k + ks) \bmod \ell \neq 0 \\
(d_2 + d_3 + \cdots + d_{k+1} + ks) \bmod \ell \neq 0 \\
\vdots \\
(d_{m-k} + \cdots + d_{m-1} + ks) \bmod \ell \neq 0 \\
\hline
\vdots \qquad \vdots \\
\hline
(d_1 + d_2 + \cdots + d_{m-1} + (m-1)s) \bmod \ell \neq 0.
\end{array}$$

After substituting the values of d_i's as indicated in the theorem statement, each inequality is of the following form,

$$(i + js) \bmod \ell \neq 0 \quad \text{where } i,j \text{ are integers.}$$

At most (q-1) consecutively subscripted variables are zero. Therefore,

$$\text{if } i = 0 \text{ then } j < q. \tag{1}$$

There are exactly $\left\lceil \frac{m-q}{q} \right\rceil$ variables with value 1 and the others have value 0, therefore

$$i \leq \left\lceil \frac{m-q}{q} \right\rceil.$$

Now $\ell \geq m$ therefore $i \leq \left\lceil \frac{\ell - q}{q} \right\rceil$ however $q | \ell$, thus $i \leq \frac{\ell - q}{q}$ which implies

$$\frac{qi}{\ell} < 1. \qquad (2)$$

Now suppose one of the constraints is violated; that is, there exists a constraint such that

$$i + js = k\ell. \qquad (3)$$

Multiplying this equation by q/ℓ we get

$$\frac{qi}{\ell} + \frac{jqs}{\ell} = qk.$$

Now $\frac{qi}{\ell}$ must be an integer since $\ell | qs$. However, $\frac{qi}{\ell} < 1$ by (2). Thus,

$$\frac{qi}{\ell} = 0 \implies i = 0.$$

Substituting in equation (3) we have $js = k\ell$. So $\ell | js$, but q is the smallest positive integer such that $\ell | qs$. Therefore

$$j \geq q.$$

But this is a contradiction to the fact that $j < q$ when $i = 0$ (by equation (1)). Therefore no constraint is violated and hence the stated assignment of d_i's makes the pipeline allowable for cycle (ℓ). Now to prove that this solution is also optimum, consider the following subset of the constraints:

$$\left.\begin{array}{l}
(d_1 + d_2 + \cdots\cdots\cdots\cdots + d_q + qs) \bmod \ell \neq 0 \\
(d_2 + d_3 + \cdots\cdots\cdots\cdots + d_{q+1} + qs) \bmod \ell \neq 0 \\
\vdots \\
(d_k + d_{k+1} + \cdots\cdots\cdots\cdots + d_{k+q-1} + qs) \bmod \ell \neq 0 \\
\vdots \\
(d_{m-q} + \cdots\cdots\cdots\cdots + d_{m-1} + qs) \bmod \ell \neq 0
\end{array}\right\} \quad (4)$$

This formulation assumes that $q < m$. If $q \geq m$ then the added delay is zero and the solution is obviously optimum. Recall that $qs \bmod \ell = 0$. Therefore to satisfy the above inequalities some variables in each inequality must be nonzero. There are $(m-q)$ inequalities in the above subset. Any single variable d_i appears in at most q of these inequalities. Therefore at least $\left\lceil \frac{m-q}{q} \right\rceil$ variables must be nonzero and the solution must have a total added delay of at least $\left\lceil \frac{m-q}{q} \right\rceil$. This is exactly the case for our choice

$$d_q = d_{2q} = \cdots\cdots = d_{\left\lceil \frac{m-q}{q} \right\rceil q} = 1, \text{ other } d_i = 0.$$

Therefore the solution is optimum. Q.E.D.

<u>Corollary 2.8.1.1</u>: If ℓ divides ms then the above solution is unique.

<u>Proof</u>: The constraints presented in the above proof can be considered as the constraints for making a single row allowable. In this case d_i should be thought of as the number of elemental delays to be added between the i^{th} and $(i+1)^{th}$ X in that row. Obviously all the rows have identical sets of constraints. Thus if a solution is unique for one row then it is also unique for any other row. The only way to apply this solution

simultaneously to all the rows in the reservation table, is to insert the delays only between iterations. This in turn shows that the complete solution is also unique. Therefore it is sufficient to show that under the given conditions the assignment of values to d_i's given by the above theorem is unique. The proof follows.

$$q = \frac{\ell}{\gcd(\ell,s)} \quad \text{by statement of the above theorem.}$$

$$\ell \mid ms \Rightarrow q\cdot\gcd(\ell,s) \mid ms \Rightarrow q \mid m\cdot\frac{s}{\gcd(\ell,s)}$$

But $q(=\frac{\ell}{\gcd(\ell,s)})$ and $\frac{s}{\gcd(\ell,s)}$ are relatively prime. Therefore $q|m$. This implies $\lceil\frac{m-q}{q}\rceil = \frac{m-q}{q}$. Then the optimum solution given by Theorem 2.8.1 is:

$$d_q = d_{2q} = \cdots\cdots = d_{(\frac{m-q}{q})q} = 1 \quad \text{and other } d_i = 0.$$

Consider again the subset (4) of inequalities used in the above proof. There are $\frac{m-q}{q}$ variables which are 1 and each one covers exactly q distinct inequalities in (4).

Since no variable appears in more than q of the constraints in (4), any other optimum assignment must have each nonzero variable also covering q distinct inequalities.

If $d_i = 1$ for some i $1 \leq i \leq q-1$, then d_i covers exactly i inequalities, which is less than q. Therefore $d_i = 1$ is not an optimum choice and therefore $d_q = 1$ in order to cover the first inequality. The first q inequalities are thus covered.

Suppose now $d_{q+i} = 1$ for some i, $1 \leq i \leq q-1$, then d_{q+i} covers q inequalities, but out of these, q-i inequalities are already covered by d_q and therefore d_{q+i} covers only $i < q$ new inequalities, and thus $d_{q+i} = 1$ is not an optimum choice. Then $d_{2q} = 1$ is needed to cover inequality q+1. The first 2q inequalities are thus covered.

In a similar manner one can prove by finite induction that $d_{3q} = d_{4q} = \cdots = d_{(\frac{m-q}{q})q} = 1$ while other d_i are 0. The assignment is thus unique and hence the complete optimum solution is also unique.

Q.E.D.

The extreme regularity of the looping pipelines and their added delays makes the sharing of noncompute segments simpler to consider, as the following theorem indicates.

Theorem 2.8.2: All the elemental delays in the solution given by Theorem 2.8.1 can share a single noncompute segment.

Proof: All the elemental delays can share a single segment if no two elemental delays are a multiple of ℓ apart in the reconfigured reservation table.

Successive elemental delays in the reconfigured table are (qs+1) apart. Therefore the distance between any two elemental delays, d_{iq} and d_{jq}, for $j > i$ is $(j-i)(qs+1)$.

Suppose this distance is a multiple of ℓ. Then $(j-i)qs + (j-i) = k\ell$ for some integer k. Dividing by ℓ we get $r + \frac{j-i}{\ell} = k$ where r is some integer, because $\ell | qs$. Thus $\frac{j-i}{\ell}$ must be an integer. We show a contradiction by proving that $(j-i)/\ell$ is a fraction. In the solution of Theorem 2.8.1 only the following delays are present,

$$d_q, d_{2q}, \ldots, d_{\lceil \frac{m-q}{q} \rceil q}.$$

Therefore

$$j \leq \left\lceil \frac{m-q}{q} \right\rceil \leq \left\lceil \frac{\ell-q}{q} \right\rceil = \frac{\ell-q}{q} = \frac{\ell}{q} - 1 < \ell.$$

Thus $\ell > j > i$, which implies $(j-i)/\ell$ is a fraction. Thus no distance between two elemental delays is a multiple of ℓ and therefore all elemental delays can share a single noncompute segment. Q.E.D.

Example 2.8.1: For the looping pipeline of Figure 2.8.1a, suppose we want to have the cycle (6) allowable. They by Theorem 2.8.1,

$$q = \frac{\ell}{\gcd(\ell,s)} = \frac{6}{\gcd(6,3)} = 2,$$

and thus the delay assignments $d_2 = d_4 = 1$ and $d_1 = d_3 = d_5 = 0$ makes the pipeline allowable for constant latency 6. By Theorem 2.8.2 d_2 and d_4 can share the same noncompute segment. The solution is given in Figure 2.8.1b where S_3 is the noncompute segment. Verify that no usage interval of Figure 2.8.1b is a multiple of 6.

2.9. Arbitrary Cycles

So far we have discussed only constant latency cycles. In this section we deal with making a pipeline allowable for any given constant or nonconstant latency cycle. We shall also present a generalization of Theorem 2.3.1. In the general case, the allowability of a pipeline cannot be judged solely from the average latency of the cycle. All we can say immediately is that for a given cycle with average latency ℓ, no pipeline with lower bound m such that $m > \ell$ can be made allowable. But what about the pipelines with $m \leq \ell$? Take for example the cycle (1,2,3). It has an average latency $\ell = 2$ and period $p = 6$. Let us try to find an allowable pipeline with lower bound $m = 2$. A pipeline with usage interval set $\underline{F}$ is allowed by cycle (1,2,3) iff the set of initiation intervals $\underline{G}$ of the cycle satisfies $\underline{F}_6 \cap \underline{G}_6 = \underline{\Phi}$. Since $\underline{G}_6$ for the cycle (1,2,3) is $\{0,1,2,3,4,5\}$, $\underline{F}_6$ must be empty for a pipeline to be allowable. Thus the allowable pipeline does not have any usage intervals, that is, each row of the reservation table has only one X. Thus the only allowable pipeline has a lower bound $m = 1$ and thus no allowable pipeline exists with lower bound $m = 2$.

Let us define a <u>maximal pipeline with respect to a cycle</u> as an allowable pipeline which has the highest lower bound among allowable pipelines.

To find the lower bound of a maximal pipeline relative to a cycle, we can construct a reservation table with as many X's as possible in some row subject to satisfying the allowability constraint, $\underline{F}_p \cap \underline{G}_p = \underline{\Phi}$. The constraint may be rewritten as $\underline{F}_p \subseteq \underline{H}_p$, where $\underline{H}_p$ is the complement of $\underline{G}_p$ in $\underline{Z}_p$ (see Section 2.2).

Two elements i and j, such that $i,j \in \underline{Z}_p$, are defined to be <u>compatible</u> if $|j-i| \in \underline{H}_p$.

Using this definition we can form all compatibility classes of the cycle. A compatibility class is one in which each element is compatible with every other element in the class.

In the theorems to follow, we establish some transformations on a row, which do not change the allowability of the row and give several ways of constructing an allowable row from a given compatibility class. Let us denote a row which has an X in each of columns $t_1, t_2, \ldots, t_k$ as row $\{t_1, t_2, \ldots, t_k\}$. Although we have used only non-negative integers for column numbers in all our examples, there is no need for such a restriction. In the following discussion column numbers are assumed to be integers without any restriction.

The following lemma is a useful redefinition of compatibility, which avoids the use of absolute quantities.

Lemma 2.9.1.1: Two integers $i, j \in Z_p$ are compatible if and only if $(i-j) \bmod p \in \underline{H}_p$.

Proof: i and j are compatible iff

$$|i-j| \in \underline{H}_p \qquad \text{(by definition).}$$

Now $|i-j| = |i-j| \bmod p$ since $0 \leq i,j < p$. Therefore we need to show that

$$|i-j| \bmod p \in H_p \iff (i-j) \bmod p \in H_p.$$

If $i > j$ then it is trivially true. Let $j > i$, then $|i-j| = (j-i)$. Hence we need to show that,

$$(j-i) \bmod p \in \underline{H}_p \iff (i-j) \bmod p \in \underline{H}_p$$

or

$$(j-i) \bmod p \in \underline{H}_p \iff p-((j-i) \bmod p) \in \underline{H}_p.$$

This we know is true from Property 2.2.3. Therefore i and j are compatible iff $(i-j) \bmod p \in \underline{H}_p$. Q.E.D.

We can make a similar simplification on the condition of allowability; that is, we show that it is not necessary to use absolute quantities.

<u>Lemma 2.9.1.2</u>: Given a pair of X's in columns t_1 and t_2 of a row, the usage interval $|t_2-t_1|$ is allowable if and only if $(t_2-t_1) \bmod p \in \underline{H}_p$.

<u>Proof</u>: By Theorem 2.2.1, the usage interval $|t_2-t_1|$ is allowable iff $|t_2-t_1| \bmod p \in \underline{H}_p$. Thus we need to show,

$$|t_2-t_1| \bmod p \in \underline{H}_p <=> (t_2-t_1) \bmod p \in \underline{H}_p.$$

If $t_2 > t_1$ then it is trivially true. Therefore let $t_1 > t_2$. Then we need to show,

$$(t_1-t_2) \bmod p \in \underline{H}_p <=> (t_2-t_1) \bmod p \in \underline{H}_p$$

or $$(t_1-t_2) \bmod p \in \underline{H}_p <=> p-[(t_1-t_2) \bmod p] \in \underline{H}_p$$

which is true by Property 2.2.3. Q.E.D.

<u>Lemma 2.9.1.3</u>: Given a set of integers $\{t_1,t_2,\ldots,t_r\}$, either all or none of the following reservation table rows are allowed by a cycle with period p.

$$\text{Row}\{[t_1+k] \bmod p + i_1 p, \ldots, [t_r+k] \bmod p + i_r p\}$$

for all integers $k,i_1,i_2,\ldots,i_r$.

<u>Proof</u>: We shall pick two arbitrary rows from the above set and show that either both or none of them are allowed by a cycle with period p, that is, we show that

$$\text{Row}\{[t_1+z] \bmod p + j_1 p, \ldots, [t_r+z] \bmod p + j_r p\} \text{ allowable}$$

$$\Leftrightarrow \text{Row}\{[t_1+z'] \bmod p + j_1' p, \ldots, [t_r+z'] \bmod p + j_r' p\} \text{ allowable.}$$

Let t_a and t_b be any two members of the set $\{t_1, \ldots, t_r\}$. By Lemma 2.9.1.2, we need to show that

$$([t_a+z] \bmod p + j_a p - [t_b+z] \bmod p - j_b p) \bmod p \in \underline{H}_p$$

$$\Leftrightarrow ([t_a+z'] \bmod p + j_a' p - [t_b+z'] \bmod p - j_b' p) \bmod p \in \underline{H}_p.$$

But we have

$$([t_a+z] \bmod p + j_a p - [t_b+z] \bmod p - j_b p) \bmod p$$

$$= ([t_a+z] - [t_b+z]) \bmod p$$

$$= ([t_a+z'] - [t_b+z']) \bmod p$$

$$= ([t_a+z'] \bmod p - [t_b+z'] \bmod p) \bmod p$$

$$= ([t_a+z'] \bmod p + j_a' p - [t_b+z'] \bmod p - j_b' p) \bmod p.$$

Hence either both rows or neither are allowable. Q.E.D.

Theorem 2.9.1: Let $\{z_1, z_2, \ldots, z_r\}$ be a compatibility class of a given cycle with period p, then all of the following rows are allowable.

$$\text{Row}\{[z_1+k] \bmod p + i_1 p, [z_2+k] \bmod p + i_2 p, \ldots, [z_r+k] \bmod p + i_r p\}$$

for all integers $k, i_1, \ldots, i_r$.

Proof: For any two elements z_a and z_b in the compatibility class we have by Lemma 2.9.1.1

$$(z_a - z_b) \bmod p \in \underline{H}_p.$$

And by Lemma 2.9.1.2 then usage interval $|z_a - z_b|$ is allowable. Hence the row $\{z_1, z_2, \ldots, z_r\}$ is allowable. By setting

$k = i_1 = i_2 = \cdots = i_r = 0$ we see that row$\{z_1, z_2, \ldots, z_r\}$ is a member of the set of rows defined in the theorem statement. By Lemma 2.9.1.3, row$\{z_1, \ldots, z_r\}$ being allowable implies that all of the rows in the theorem statement are allowable. Q.E.D.

Sometimes we may want to use only very simple transformations on a compatibility class to obtain an allowable row. The following corollary gives three such transformations.

Corollary 2.9.1.1: Let $\{z_1, z_2, \ldots, z_r\}$ be a compatibility class of a cycle with period p, then the following rows are allowable:

(a) Row$\{z_1+k, z_2+k, \ldots, z_r+k\}$ $\forall$ integers k

(b) Row$\{(z_1+k) \bmod p, \ldots, (z_r+k) \bmod p\}$ $\forall$ integers k

(c) Row$\{z_1+i_1 p, \ldots, z_r+i_r p\}$ $\forall$ integers $i_1, \ldots, i_r$.

Proof: The sets of rows in (a), (b) and (c) can be shown to be subsets of the sets of rows defined in Theorem 2.9.1. Hence the rows in (a), (b) and (c) are allowable. Q.E.D.

We might point out here that the union of sets in (a), (b) and (c) of the above corollary does not necessarily equal the set defined in Theorem 2.9.1, but if (b) is applied to a given row, $\{z_1, z_2, \ldots, z_r\}$, and (c) is applied to the resulting set of rows, the resulting set is equal to the set defined in Theorem 2.9.1.

We might ask the question whether the procedure given in the Theorem 2.9.1 is sufficient to construct all allowable rows, in the sense that if the procedure is applied to all the compatibility classes of a cycle, all possible rows allowed by the given cycle will be generated. The following lemma shows that even the simple operation (c) in Corollary 2.9.1.1 is sufficient.

Lemma 2.9.2.1: Given a cycle with period p and the set $\underline{C}$ consisting of all its compatibility classes, the following rows are the only rows which are allowed by the cycle:

$$\text{Row } \{z_1+i_1p, z_2+i_2p, \dots\}$$

for all integers $i_1, i_2, \dots$ and for all $\{z_1, z_2, \dots\} \in \underline{C}$.

Proof: From (c) of Corollary 2.9.1.1, the above rows are allowable. Next we show that every allowable row is included in the above set of rows. Let row$\{t_1, t_2, \dots, t_k\}$ be an allowable row. They by Lemma 2.9.1.2, for any two elements t_a and t_b of the row

$$(t_a - t_b) \bmod p \in \underline{H}_p$$

let

t_i be expressed as $(t_i \bmod p) + j_i p$ for $1 \le i \le k$. (1)

Then $(t_a - t_b) \bmod p \in \underline{H}_p$

$$\Rightarrow ([t_a \bmod p] + j_a p - [t_b \bmod p] - j_b p) \bmod p \in \underline{H}_p$$

$$\Rightarrow ([t_a \bmod p] - [t_b \bmod p]) \bmod p \in \underline{H}_p.$$

We also have that

$$(t_a \bmod p) \in \underline{Z}_p \quad \text{and} \quad (t_b \bmod p) \in \underline{Z}_p.$$

Therefore by Lemma 2.9.1.1 $(t_a \bmod p)$ and $(t_b \bmod p)$ are compatible and hence $\{(t_1 \bmod p), \dots, (t_k \bmod p)\}$ is a compatibility class. Then by (c) of Corollary 2.9.1.1

$$\text{row}\{(t_1 \bmod p) + j_1 p, \dots, (t_k \bmod p) + j_k p\}$$

is allowable and included in the set of rows defined in the statement of this lemma. By (1) this row is identical to the row $\{t_1,\ldots,t_k\}$ and hence row $\{t_1,\ldots,t_k\}$ is included in the set defined by this lemma. Q.E.D.

Theorem 2.9.2: The lower bound of a maximal pipeline with respect to a given cycle, is equal to the size of the largest compatibility class of the cycle.

Proof: By Lemma 2.9.2.1, the only rows which are allowed by a cycle with period p are:

$$\text{Row}\{z_1+i_1p, z_2+i_2p, \ldots\}$$

for all integers $i_1, i_2, \ldots$ and for all compatibility classes $\{z_1, z_2, \ldots\}$ of the cycle. Thus the maximum number of X's in any allowable row is equal to the size of the largest compatibility class. Thus the lower bound of a maximal pipeline is equal to the size of the largest compatibility class. Q.E.D.

Now to gain information regarding the lower bound of a maximal pipe or the set of allowable rows with respect to a cycle we need to form only the maximal compatibility classes of the cycle. A maximal compatibility class is a compatibility class which is not a subset of any other compatibility class.

We can further limit ourselves to producing only those maximal compatibility classes which contain element 0, because the other classes can always be produced by adding a constant to the members of some compatibility class containing 0, furthermore these classes (classes not containing 0) do not yield any new information on allowability of rows. This fact is formally stated in the following theorem.

<u>Theorem 2.9.3</u>: Let $\{z_1, z_2, \ldots, z_i, \ldots, z_r\}$ be a compatibility class of some given cycle, not containing 0. Then there exists a compatibility class containing 0, such that the use of Theorem 2.9.1 on both these classes results in two identical sets of allowable rows. This class is $\{z_1 - z_i, z_2 - z_i, \ldots, z_i - z_i, \ldots, z_r - z_i\}$, where z_i is the smallest element of the given class.

<u>Proof</u>: For any two elements z_a and z_b of the given class

$$|z_a - z_b| \in \underline{H}_p \quad \text{(by definition of compatibility).}$$

Also given that,

$$z_a, z_b, z_i \in \underline{Z}_p \quad \text{and} \quad z_a, z_b \geq z_i,$$

it follows that

$$(z_a - z_i) \in \underline{Z}_p \quad \text{and} \quad (z_b - z_i) \in \underline{Z}_p.$$

Thus $(z_a - z_i)$ and $(z_b - z_i)$ are compatible and hence $\{z_1 - z_i, z_2 - z_i, \ldots, z_i - z_i, \ldots, z_r - z_i\}$ is a compatibility class containing 0. The sets of allowable rows defined by Theorem 2.9.1 on $\{z_1, \ldots, z_r\}$ and $\{z_1 - z_i, \ldots, z_r - z_i\}$ are clearly identical because k of Theorem 2.9.1 takes all integer values. Q.E.D.

Here are some examples.

<u>Example 2.9.1</u>: Cycle (1,9), period $p = 10$.

$$\underline{G}_{10} = \{0,1,9\} \quad , \; \underline{H}_{10} = \{2,3,4,5,6,7,8\}.$$

Maximal compatibility classes containing 0 are:

$\{0,2,4,6,8\},\{0,2,4,7\},\{0,2,5,7\},\{0,2,5,8\},$

$\{0,3,5,7\},\{0,3,5,8\},\{0,3,6,8\}.$

The cardinality of the largest class is 5, therefore by Theorem 2.9.2 the lower bound of the maximal pipeline with respect to cycle (1,9) is 5. In other words no allowable row can have more than 5 X's. □

For the hand calculation of maximal compatibility classes containing 0, we suggest the following systematic procedure. For a better understanding refer to Figure 2.9.1 when reading the following procedure. Figure 2.9.1 corresponds to the generation of maximal compatibility classes of the cycle (1,9) of the above example.

Algorithm 2.9.1:

C1. Form the set $\underline{H}_p$ of the cycle by the usual procedures. Write 0 at the root of the tree, which is to be generated.

C2. Compute the immediate successors of 0 and place them in increasing order from left to right. Where the immediate successors of 0 are all the integers which are compatible with 0 (these are simply the elements of $\underline{H}_p$).

C3. Pick the leftmost leaf-node in the tree which is not circled or crossed out; call this node N. If no such nodes exist, then terminate this procedure. Each set of nodes on the path from the root of the tree to a circled node corresponds to a maximal compatibility class.

C4. Generate the immediate successors of node N and place them in increasing order from left to right. The immediate successors of

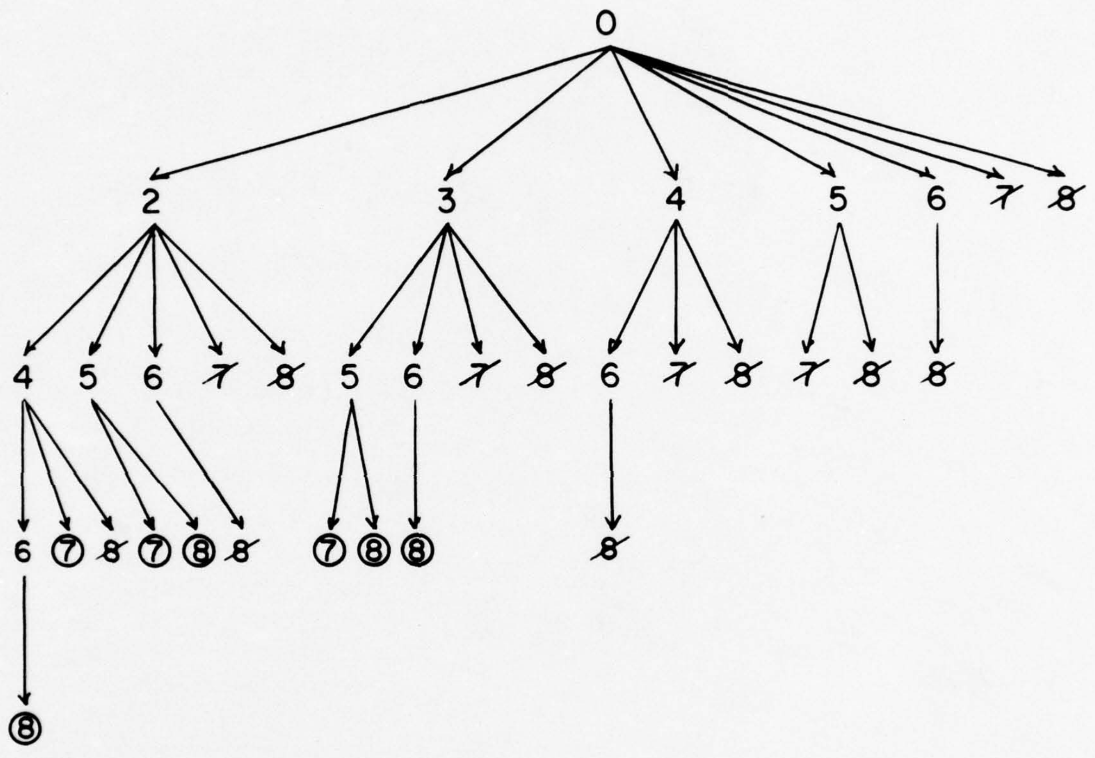


Figure 2.9.1. Generating maximal compatibility classes of cycle (1,9) using Algorithm 2.9.1.

N are defined to be those brothers of N which appear to the right of N and whose values are compatible with the value of N. Nodes which have the same immediate predecessor are brothers. If no such successor exists then check if this compatibility class (the set of nodes on the path from the root of the tree to node N) is a subset of a previously generated compatibility class (the set of nodes on the path from the root of the tree to a circled node). If not a subset then circle the node N else cross it out. Go to step C3. □

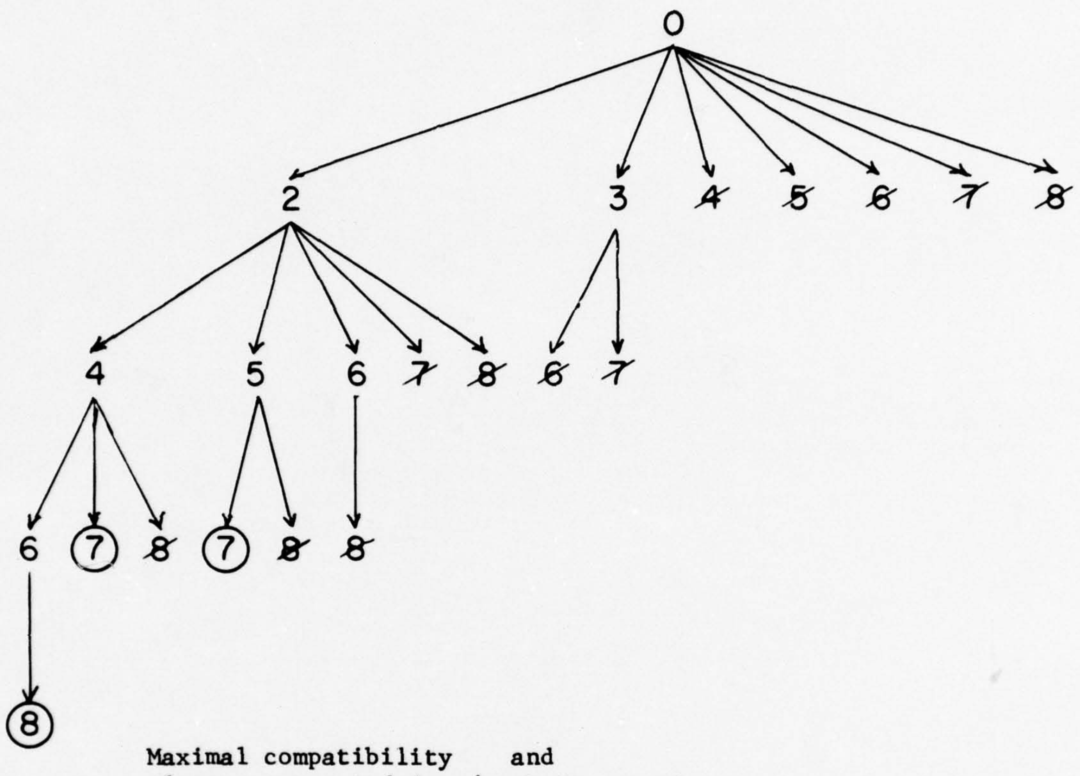
The above algorithm works in the following manner. For each node of the tree we generate all its successors which are compatible with it and all its predecessors. Thus a node and its brothers are the only elements which are compatible with the node's predecessors. Therefore to generate a node's successors, one only needs to look at the brothers of that node. When we no longer have a successor to a node we have generated a maximal compatibility class or a subset of an already generated maximal compatibility class. The above algorithm can be speeded up by taking into account the special nature of compatibility classes. This is discussed below.

By Corollary 2.9.1.1b if $\{z_1,\ldots,z_r\}$ is an allowable row of a cycle then for any integer k, $\{(z_1+k) \bmod p,\ldots,(z_r+k) \bmod p\}$ is also an allowable row. The effect of adding k modulo p is to rotate the X's of the row. That is, if one considers a row with p cells and an X each in cell $z_1,z_2,\ldots,z_r$ then $\{(z_1+k) \bmod p,\ldots,(z_r+k) \bmod p\}$ is a row obtained by end-around shift of X's by k cells. The same rotation operation is also applicable to compatibility classes since a compatibility class is also an allowable row and an allowable row is also a compatibility class if the elements of the row are between 0 and p-1. Thus, if $\{0,z_1,\ldots,z_i,\ldots\}$ is a compatibility class containing 0 then $\{(0+z_i') \bmod p, (z_1+z_i') \bmod p, \ldots, (z_i+z_i') \bmod p, \ldots\}$ is also a compatibility class containing 0, where $z_i' = p-z_i$. Therefore, if given all the maximal compatibility classes containing say 0 and z, one can generate all maximal classes containing 0 and p-z by a rotation operation. Moreover, one can also generate by rotation, all classes which contain two elements with difference z; i.e., all classes of the type $\{0,\ldots,i,i+z,\ldots\}$. These two properties allow us the following algorithm. In this algorithm as soon as we form a maximal compatibility class we also form all its rotations. Once all the classes containing, say i, and their rotations are formed, we eliminate (p-i) from the tree and restrict ourselves to forming compatible pairs with a difference strictly greater than i. The tree formed by the following algorithm for cycle (1,9) is given in Figure 2.9.2. Compare it with Figure 2.9.1.

<u>Algorithm 2.9.2</u>:

C1. Form the set $\underline{H}_p$ of the cycle by the usual procedures. Write 0 at the root of the tree, which is to be generated.

C2. Compute the immediate successors of 0 and place them in increasing order from left to right. Where the immediate successors of 0 are all the integers which are compatible with 0 (these are simply the elements of $\underline{H}_p$).



Maximal compatibility classes generated in the tree	:	and their rotations
{0,2,4,6,8}	:	
{0,2,4,7}	:	{0,3,5,7},{0,3,6,8},{0,2,5,8}
{0,2,5,7}	:	{0,3,5,8}

Figure 2.9.2. Generating maximal compatibility classes of cycle (1,9) using Algorithm 2.9.2.

C3. Pick the leftmost leaf-node in the tree,

call this node N;

$i \leftarrow$ value of N.

C4. Generate the immediate successors of node N and place them in increasing order from left to right. The immediate successors of N are defined to be those brothers of N (which are not crossed out by C5) whose values are at least i +value(N) and are compatible with the value of N. Nodes which have the same immediate predecessor are brothers. If no such successor exists then check if this compatibility class (the set of numbers on the path from the root of the tree to node N) is a subset of a previously generated compatibility class. If it is a subset then cross out node N else circle the node N and generate other maximal classes by rotation.

C5. Pick the leftmost leaf-node in the tree which is not circled or crossed out; call it N. If no such node exists then terminate this procedure. If N is an immediate successor of 0 then cross out the rightmost leaf-node which is not yet crossed out, and set $i \leftarrow$ value(N). Go to C4. □

Note that the compatibility classes formed directly in the tree (i.e. not by rotation) are sufficient to generate all allowable rows by the use of Theorem 2.9.1, since the theorem already incorporates the rotation operation. Thus in the example of Figure 2.9.2 the following classes are sufficient to characterize all the allowable rows of cycle (1,9):

$$\{0,2,4,6,8\},\{0,2,4,7\},\{0,2,5,7\}.$$

One more example follows.

Example 2.9.2: For cycle (2,3,7), period $p = 12$,

$$\underset{\sim}{G}_{12} = \{0,2,3,5,7,9,10\}, \quad \underset{\sim}{H}_{12} = \{1,4,6,8,11\}.$$

All maximal compatibility classes containing 0 are:

$$\{0,1\},\{0,4,8\},\{0,6\},\{0,11\}.$$

Of these the first three classes are sufficient to characterize all the allowable rows by use of Theorem 2.9.1. The size of the largest class is 3, hence the lower bound of the maximal pipeline is 3 (by Theorem 2.9.2). This means that no pipeline with lower bound 4 is allowed by the cycle (2,3,7), even though the average latency of the cycle is 4. This also means that if cycle (2,3,7) is used as an initiation cycle on some allowable pipeline then no segment is used more than $3/4 = 75\%$ of the time. Thus the cycle (2,3,7) is imperfect in some sense. □

Let us define a cycle to be perfect if the lower bound of its maximal pipeline equals the average latency of the cycle. Thus the cycle (1,9) of Example 2.9.1 is perfect. There is no straightforward method to test the perfectness of a cycle. However, the following theorem gives a method to generate a large number of perfect cycles.

Lemma 2.9.4.1: If none of the nonzero elements of $\underset{\sim}{G}_p$ of a cycle $(i_0, i_1, \ldots, i_{n-1})$ is divisible by n, then $n|p$ and the cycle is a perfect cycle.

Proof: Let $\ell = \lceil \frac{p}{n} \rceil$. We show that $\{0,n,2n,\ldots,(\ell-1)n\}$ is a compatibility class of the cycle. First we must show that the elements of the class are in $\underset{\sim}{Z}_p$. All elements are nonnegative, therefore it is sufficient to show that $(\ell-1)n < p$.

$$\ell = \lceil \frac{p}{n} \rceil \text{, therefore } \ell < \frac{p}{n} + 1 \quad \text{hence } (\ell-1)n < p.$$

Now suppose $\{0,n,\ldots,(\ell-1)n\}$ is not a compatibility class. Then $\exists$ two elements m_1n and m_2n in the above class such that $0 \leq m_1 < m_2 \leq (\ell-1)$ and $m_2n - m_1n \notin \underset{\sim}{H}_p$. Therefore $(m_2-m_1)n \in \underset{\sim}{G}_p$. Now $m_1 \neq m_2$. Therefore $(m_2-m_1)n$ is a nonzero element of $\underset{\sim}{G}_p$ which is divisible by n, a contradiction. Therefore $\{0,n,2n,\ldots,(\ell-1)n\}$ is a compatibility class with ℓ elements. This implies that the given cycle with average latency p/n is allowed by some pipeline with lower bound ℓ. Since no segment can be used more than 100% of the time we must have, $\ell \leq \frac{p}{n}$. However, $\ell = \lceil \frac{p}{n} \rceil \Rightarrow \ell \geq \frac{p}{n}$. Therefore $\ell = p/n$. Thus $n|p$ and the cycle is perfect. Q.E.D.

Theorem 2.9.4: If none of the nonzero elements of $\underset{\sim}{G}_p$ of a cycle $(i_0,i_1,\ldots,i_{n-1})$ is divisible by n, then the following cycles are perfect:

$$\text{cycle}((i_0+j_0n)k,(i_1+j_1n)k,\ldots,(i_{n-1}+j_{n-1}n)k)$$

$$\forall \text{ integers } j_0,j_1,\ldots,j_{n-1} \geq 0 \text{ and } k \geq 1.$$

Proof: Let $\underset{\sim}{G}'$ be the set of initiation intervals of cycle $((i_0+j_0n)k,\ldots,(i_{n-1}+j_{n-1}n)k)$ and p' be its period. Clearly $n|p'$, since by Lemma 2.9.4.1 $n|(i_0+i_1+\cdots+i_{n-1})$. Let $p'/n = \ell\cdot k$. Then we show that

$$\{0,1,2,\ldots,(k-1),nk,nk+1,\ldots,nk+(k-1),\ldots,(\ell-1)nk,$$
$$(\ell-1)nk+1,\ldots,(\ell-1)nk+(k-1)\}$$

is a compatibility class with ℓk elements and hence the cycle is perfect.

Suppose the above is not a compatibility class, then there exist two elements m_1nk+m_3 and m_2nk+m_4 in the above class such that $m_1nk+m_3 < m_2nk+m_4$ and $0 \leq m_1,m_2 \leq (\ell-1)$, $0 \leq m_3,m_4 \leq (k-1)$ and $(m_2nk+m_4) - (m_1nk+m_3) \in \underline{G}'$ mod p'.

All the nonzero elements of $\underline{G}'_{p'}$ are of the form $(g+mn)k$, where $g \neq 0$ and $g \in \underline{G}_p$ and integer $m \geq 0$. Therefore, $(m_2nk+m_4) - (m_1nk+m_3) = (g+mn)k$. Dividing by k, we get

$$m_2n - m_1n + (m_4-m_3)/k = g+mn.$$

Since $0 \leq m_3,m_4 \leq k-1$, if $(m_4-m_3)/k$ is to be an integer, then (m_4-m_3) must be zero. Thus,

$$m_2n - m_1n = g+mn$$

or

$$m_2-m_1 = g/n + m.$$

This is a contradiction, since $g \neq 0$ and n does not divide g. Therefore the assumed class is a compatibility class of the cycle $((i_0+j_0n)k,\ldots,(i_{n-1}+j_{n-1}n)k)$ and hence the cycle is perfect. Q.E.D.

A large number of perfect cycles can be generated using the above theorem, provided one already has a cycle which satisfies the nondivisibility condition of the above theorem. The following corollary uses a very simple condition to generate a set of perfect cycles.

Corollary 2.9.4.1: If i and n are relatively prime (i.e., gcd(i,n) = 1) then the following cycles are perfect:

$$\text{cycles}((i+j_0n)k, (i+j_1n)k, \ldots, (i+j_{n-1}n)k)$$

$$\forall \text{ integers } j_0, j_1, \ldots, j_{n-1} \geq 0 \text{ and } k \geq 1.$$

Proof: $\underline{G}_p$ of cycle $(i,i,\ldots,i)$ is $\{0,i,2i,\ldots,(n-1)i\}$ and hence no nonzero element of $\underline{G}_p$ is divisible by n. Then by Theorem 2.9.4 the above cycles are perfect. Q.E.D.

The following is a direct consequence of the above corollary with n set to 1.

Corollary 2.9.4.2: All constant latency cycles are perfect. □

Example 2.9.3: gcd(2,3) = 1, therefore by Corollary 2.9.4.1 cycle (2,2+3,2+6) = (2,5,8) is a perfect cycle. From the proof of Theorem 2.9.4, one of the largest compatibility classes is $\{0,3,6,9,12\}$.

Cycle $(2\times2, 5\times2, 8\times2) = (4,10,16)$ is also perfect. One of its compatibility classes having $(4+10+16)/3 = 10$ elements is, from Theorem 2.9.4, $\{0,1,6,7,12,13,18,19,24,25\}$. □

Example 2.9.4: For cycle (2,1,2,3), $\underline{G}_8 = \{0,1,2,3,5,6,7\}$. The number of initiations in the cycle is $n = 4$. None of the positive elements of $\underline{G}_8$ is divisible by 4. Therefore cycle (2,1,2,3) is perfect and so are:

$$\text{cycles } (2+4i_0, 1+4i_1, 2+4i_2, 3+4i_3) \quad \forall \text{ integers } i_0, i_1, i_2, i_3 \geq 0. \quad \square$$

Next we turn our attention to making a pipeline allowable by use of noncompute segments. The following theorem is a generalization of Theorem 2.3.1.

Theorem 2.9.5: For a given cycle with period p, a pipeline can be made allowable using noncompute segments if and only if the lower bound b of the maximal pipeline of the cycle is greater than or equal to the lower bound m of the pipeline.

Proof: The 'only if' part is trivially true from the definition of the maximal pipeline, namely, that no allowable pipeline has lower bound greater than b.

If $b \geq m$ then the pipeline can be made allowable as shown below, by making every row of the reservation table of the form described in Lemma 2.9.2.1, that is, every row is of the form

$$\mathrm{row}\{z_1+i_1p, z_2+i_2p, \ldots\ldots\}$$

for some compatibility class $\{z_1, z_2, \ldots\}$ of the given cycle and for some integers $i_1, i_2, \ldots$

To achieve this form, scan the columns of the given reservation table from left to right, scan each column from top to bottom for X's. An X may be moved to the right to ensure that the column in which this X appears has a number not equivalent modulo p to that of any previously scanned X in the same row; and to ensure that all these column numbers modulo p (in the same row) are contained in some single compatibility class of the cycle. This can always be done because the largest compatibility class has b elements, (by definition of maximal pipeline) and there are no more than m X's in any row, where $m \leq b$.

As described in the proof of Theorem 2.3.1, moving an X to the right in the reservation table is equivalent to inserting appropriate noncompute delays in the pipeline. Q.E.D.

Example 2.9.5: We shall illustrate the construction of the above proof on the reservation table of Figure 2.9.3a. Its lower bound is $m = 2$. Suppose we want to make this pipeline allowable for cycle (2,3,4). First we verify that the lower bound of the maximal pipeline relative to the cycle (2,3,4) is greater than or equal to 2. The cycle has period $p = 9$,

$$\underline{G}_9 = \{0,2,3,4,5,6,7\} \quad \text{and} \quad \underline{H}_9 = \{1,8\}.$$

Maximal compatibility classes containing element 0 are, $\{0,1\}$ and $\{0,8\}$. Therefore the lower bound of the maximal pipeline is 2. Figure 2.9.3b is a reconfigured table which is allowable. Verify that the forbidden set $\underline{F}$ is $\{8,10\}$ and therefore

$$\underline{F}_9 = \{1,8\} \subseteq \underline{H}_9 \text{ as required.}$$ □

Next we consider the sharing of noncompute segments by the elemental delays. We shall assume that during one time unit a noncompute segment can provide exactly one elemental delay. The definition for the incompatibility between two elemental delays is similar to that for constant latency cycles in Section 2.4; that is, the elemental delays d_i and d_j are incompatible iff $|t_i - t_j| \bmod p \in \underline{G}_p$. Where t_i and t_j are the labels of the columns in which d_i and d_j appear

	0	1	2	3	4
S_0	X		X		
S_1		X			X
S_2				X	

(a) Starting reservation table.

	0	1	2	3	4	5	6	7	8	9	10	11
S_0	X		d_1	d_2	d_3	d_4	d_5	d_6	X			
S_1		X									d_7	X
S_2										X		

(b) Pipeline (a) made allowable.

	0	1	2	3	4	5	6	7	8	9	10	11	
S_0	X								X				
S_1		X										X	
S_2										X			
S_3			X								X		$\{d_1, d_7\}$
S_4				X	X								$\{d_2, d_3\}$
S_5						X							$\{d_4\}$
S_6							X	X					$\{d_5, d_6\}$

(c) Assignment of elemental delays to noncompute segments.

<u>Figure 2.9.3</u>. Making a pipeline allowable with respect to cycle (2,3,4).

in the reconfigured reservation table. Forming the incompatibility classes is not very useful because these classes are not disjoint, since the incompatibility relation is not transitive as it was for constant latency cycles. Conversely we define d_i and d_j to be compatible if $|t_i - t_j| \bmod p \in \underline{H}_p$. We form all the maximal compatibility classes of the elemental delays. The elements of a compatibility class can share a single noncompute segment. Now the problem reduces to the standard covering problem, that is, finding the minimum number of compatibility classes which cover all the elemental delays.

Consider the previous example. Figure 2.9.3b shows the elemental delays d_1 through d_7. For the given cycle (2,3,4), $\underline{H}_9$ is $\{1,8\}$. Then the maximal compatibility classes of elemental delays are:

$$\{d_1,d_2\},\{d_2,d_3\},\{d_3,d_4\},\{d_4,d_5\},\{d_5,d_6\},\{d_1,d_7\},$$

One of many possible minimal covers is $\{d_1,d_7\},\{d_2,d_3\},\{d_4,d_5\},\{d_5,d_6\}$. Thus 4 noncompute segments are required to provide all the elemental delays. When the maximal compatibility classes of a minimal cover are assigned to noncompute segments, elemental delays appearing in more than one maximal compatibility class are deleted from all compatibility classes but one, to obtain a disjoint set of compatibility classes. This set of compatibility classes now forms a minimal disjoint cover. One such possible assignment is shown in Figure 2.9.3c, where S_3, S_4, S_5 and S_6 are noncompute segments.

Of course the same techniques could have been used in the case of constant latency cycles. However, the techniques described in

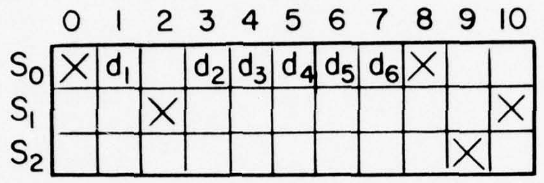
Section 2.4 are simpler to use for constant latency cycles. Take the example of Figure 2.4.1 discussed in Section 2.4. There were 4 incompatibility classes of the elemental delays d_1 through d_{10}, instead if we generated compatibility classes, there would have have been 36 of them and finding a minimal cover would have required considerable effort.

Our next concern is making a pipeline allowable for a non-constant latency cycle with minimum added execution delay. The problem formulation is identical to the constant latency problem described in Section 2.6. In principle the constraints are still the same, namely $\underline{F}_p \cap \underline{G}_p = \underline{\Phi}$. The only difference being that $\underline{G}_p$ is no longer always equal to $\{0\}$ and therefore the constraints 2.6.2 are changed to read,

$$(d'_{ab} + \sum_{b<j<c} \max_{0\leq i<I} (d_{ij}) + d_{ac} + (c-b)) \bmod p \notin \underline{G}_p \tag{2.9.1}$$

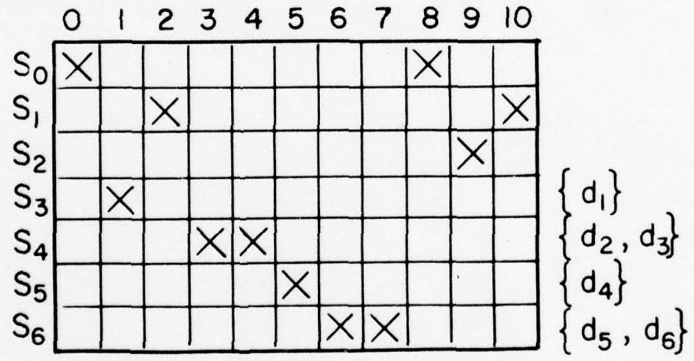
for each pair $\langle X_{ab}, X_{ac}\rangle$ with $c > b$. In the above, "$\in H_p$" may be substituted for "$\notin G_p$". The objective function, D, is unchanged. The algorithm to obtain an optimum solution is also identical to the one presented in Section 2.7. Before executing the algorithm it is not necessary to determine whether the pipeline can be made allowable by comparing the lower bound of the given pipeline to that of the maximal pipeline relative to the cycle. The algorithm terminates without a solution if no solution exists.

Continuing with Example 2.9.5, an optimum solution to make the pipeline of Figure 2.9.3a allowable to cycle (2,3,4) is given in



	0	1	2	3	4	5	6	7	8	9	10
S_0	X	d_1		d_2	d_3	d_4	d_5	d_6	X		
S_1			X								X
S_2										X	

(a) Minimum delay solution to pipeline of Fig. 2.9.3a.



	0	1	2	3	4	5	6	7	8	9	10	
S_0	X								X			
S_1			X								X	
S_2										X		
S_3		X										$\{d_1\}$
S_4				X	X							$\{d_2, d_3\}$
S_5						X						$\{d_4\}$
S_6							X	X				$\{d_5, d_6\}$

(b) Optimum assignment of delays.

<u>Figure 2.9.4</u>. Making a pipeline allowable with respect to cycle (2,3,4).

Figure 2.9.4a. An optimum assignment of elemental delays to noncompute segments for that solution is given in Figure 2.9.4b.

2.10. Multifunction Pipelines

Most of the results concerning single function pipelines can be extended to multifunction pipelines. In this section we shall present most of these results without proof. Even though the notational complexity increases considerably, the concepts still remain straightforward.

Let $\underline{F}_{XY}$ be the set of usage intervals of all $\langle X,Y \rangle$ pairs in the reservation table. This set can be formed from the reservation table by taking all pairwise distances between an X and a Y which appears to the right of the X in the same row. If both X and Y appear in the same cell of the reservation table then the pairwise distance is considered to be zero. An example is shown in Figure 2.10.1.

Similarly we define $\underline{G}_{XY}$, the set of initiation intervals of all $\langle X,Y \rangle$ pairs of a cycle, to be the set which contains all intervals of an operand of type X from a previously initiated operand of type Y.

Note that X and Y are used as variables in the definitions and in the following discussion. They take on values from the set of function types; the values need not be distinct. The following are generalizations of Properties 2.2.1, 2.2.2 and 2.2.3.

	0	1	2	3	4
S_0	A	B		A	B
S_1		A		B	
S_2	B		AB		A

Sets of usage intervals

$\underline{F}_{AA} = \{2,3\}$

$\underline{F}_{BA} = \{0,2,4\}$

$\underline{F}_{AB} = \{0,1,2,4\}$

$\underline{F}_{BB} = \{2,3\}$

<u>Figure 2.10.1</u>. Usage intervals of a multifunction pipeline.

<u>Property 2.10.1</u>: a. $0 \in \underline{G}_{XX} \bmod p$ and $ip \in \underline{G}_{XX}$ $\forall\ i \geq 1$ always.

b. If $g \neq 0$ then $g \in \underline{G}_{XX} \bmod p \Rightarrow g + ip \in \underline{G}_{XX}$ $\forall\ i \geq 0$.

c. If $X \neq Y$ then $g \in \underline{G}_{XY} \bmod p \Rightarrow g + ip \in \underline{G}_{XY}$ $\forall\ i \geq 0$. □

<u>Property 2.10.2</u>: a. $0 \in \underline{G}_{XY} \bmod p \Longleftrightarrow 0 \in \underline{G}_{YX} \bmod p$.

b. If $g \neq 0$ then $g \in \underline{G}_{XY} \bmod p \Longleftrightarrow (p-g) \in \underline{G}_{YX} \bmod p$. □

<u>Example 2.10.1</u>: Cycle $(4_A, 2_B, 5_A, 4_C, 0_B)$, period $p = 15$.

$\underline{G}_{AA} \bmod 15 = \{0,7,8\}$	$\underline{G}_{BA} \bmod 15 = \{2,4,10,11\}$	$\underline{G}_{CA} \bmod 15 = \{4,11\}$
$\underline{G}_{AB} \bmod 15 = \{4,5,11,13\}$	$\underline{G}_{BB} \bmod 15 = \{0,6,9\}$	$\underline{G}_{CB} \bmod 15 = \{0,9\}$
$\underline{G}_{AC} \bmod 15 = \{4,11\}$	$\underline{G}_{BC} \bmod 15 = \{0,6\}$	$\underline{G}_{CC} \bmod 15 = \{0\}$ □

Let $\underline{H}_{XY}$ be the complement of the set $\underline{G}_{XY}$ in the set of nonnegative integers. Then

$$\underline{H}_{XY} \bmod p = \underline{Z}_p - (\underline{G}_{XY} \bmod p).$$

<u>Property 2.10.3</u>: a. $0 \in \underline{H}_{XY} \bmod p \Longleftrightarrow 0 \in \underline{H}_{YX} \bmod p$.

b. If $h \neq 0$ then $h \in \underline{H}_{XY} \bmod p \Longleftrightarrow (p-h) \in \underline{H}_{YX} \bmod p$. □

Having defined $\underline{F}_{XY}$, $\underline{G}_{XY}$ and $\underline{H}_{XY}$, we state the following theorem which is a generalization of Theorem 2.2.1.

<u>Theorem 2.10.1</u>: A cycle with period p is allowed by a multifunction pipeline if and only if

$$(\underline{F}_{XY} \bmod p) \cap (\underline{G}_{XY} \bmod p) = \underline{\Phi}$$

or equivalently if and only if

$$(\underline{F}_{XY} \bmod p) \subseteq \underline{H}_{XY} \bmod p$$

$\forall\ X,Y \in$ the set of function types present in the cycle. □

Note that the set of function types need contain only those names which appear in a cycle rather than all function types which appear in the reservation table.

Let us represent a row by the set of integers with subscripts, where the integers are column numbers and subscripts are function names. Thus for example a row $\{4_A, 2_B, 2_C, 7_A, 5_C\}$ means that A appears in columns 4 and 7, B appears in column 2 and C appears in columns 2 and 5 of that row.

Given a cycle with period p, compatibility is defined as follows:

Two elements i_X and j_Y, such that $i,j \in \underline{Z}_p$ and $X,Y \in$ the set of function types are <u>*compatible*</u> *if* for $j \geq i$ $(j-i) \in \underline{H}_{XY} \bmod p$ (or if for $i \geq j$ $(i-j) \in \underline{H}_{YX} \bmod p$).

The following lemma follows from Property 2.10.3 and gives a convenient way to test compatibility, which avoids the comparison of i and j. This is a generalization of Lemma 2.9.1.1.

<u>Lemma 2.10.2.1</u>: Two elements i_X and j_Y, such that $i,j \in \underline{Z}_p$ and $X,Y \in$ the set of function types, are compatible if and only if $(j-i) \bmod p \in \underline{H}_{XY} \bmod p$. □

The following are generalizations of Lemmas 2.9.1.2, 2.9.1.3, Theorem 2.9.1, and Corollary 2.9.1.1 respectively.

<u>Lemma 2.10.2.2</u>: Given a pair of functions $\langle X,Y \rangle$ in columns $\langle t_1, t_2 \rangle$ of a row, the usage interval is allowable if and only if

$$(t_2 - t_1) \bmod p \in \underline{H}_{XY} \bmod p.$$ □

Lemma 2.10.2.3: Given a set $\{i_X, j_Y, \ldots..\}$ either all or none of the following rows are allowed by a cycle with period p.

$$\text{Row } \{[(i+k) \bmod p + i_1 p]_X, [(j+k) \bmod p + j_1 p]_Y, \ldots.. \}$$

$$\forall \text{ integers } k, i_1, j_1, \ldots \qquad \square$$

Theorem 2.10.2: Let $\{i_X, j_Y, \ldots..\}$ be a compatibility class of a given cycle with period p then all of the following rows are allowable.

$$\text{Row } \{[(i+k) \bmod p + i_1 p]_X, [(j+k) \bmod p + j_1 p]_Y, \ldots...\}$$

$$\forall \text{ integers } k, i_1, j_1, \ldots.. \qquad \square$$

Corollary 2.10.2.1: Let $\{i_X, j_Y, \ldots\}$ be a compatibility class of a given cycle with period p, then the following rows are allowable

(a) row $\{[i+k]_X, [j+k]_Y, \ldots. \}$ $\forall$ integers k

(b) row $\{[(i+k) \bmod p)]_X, [(j+k) \bmod p]_Y, \ldots. \}$ $\forall$ integers k

(c) row $\{[i+i_1 p]_X, [j+j_1 p]_Y, \ldots. \}$ $\forall$ integers $i_1, j_1, \ldots$ $\square$

The following lemma (generalization of Lemma 2.9.2.1) establishes the sufficiency of the procedure of Theorem 2.10.2 in forming allowable rows.

Lemma 2.10.3.1: Given a cycle with period p and the set $\underline{C}$ consisting of all its compatibility classes, the following rows are the only rows which are allowed by the cycle.

$$\text{Row } \{[i+i_1 p]_X, [j+j_1 p]_Y, \ldots..\}$$

$$\forall \text{ integers } i_1, j_1, \ldots \text{ and } \forall \{i_X, j_Y, \ldots\} \in \underline{C}. \qquad \square$$

Now we form the maximal compatibility classes from a given cycle. The maximal compatibility classes are all we need to form any allowable row. As before we need form only those classes which contain at least one 0 element; i.e., 0_X for some function X. To generate maximal compatibility classes we may use Algorithm 2.9.1 with slight modification. First of all, the "greater than" relation should be properly defined. Let us call i_X greater than j_Y if $i > j$. If $i = j$ then any fixed order can be imposed on the function types, e.g., we might say that $i_A < i_B < i_C \ldots$, etc. Next, we must generate one tree for each function type with roots 0_A, 0_B, 0_C, ..., etc. To avoid duplication, 0_X is precluded from appearing in any tree with 0_Y as a root if $0_Y > 0_X$. The following example will illustrate these changes.

Example 2.10.2: Cycle $(1_A, 1_B, 2_A)$ period $p = 4$.

The sets of initiation intervals modulo p	The sets of allowable usage intervals modulo p
$\underline{G}_{AA} \bmod 4 = \{0,1,3\}$	$\underline{H}_{AA} \bmod 4 = \{2\}$
$\underline{G}_{AB} \bmod 4 = \{2,3\}$	$\underline{H}_{AB} \bmod 4 = \{0,1\}$
$\underline{G}_{BA} \bmod 4 = \{1,2\}$	$\underline{H}_{BA} \bmod 4 = \{0,3\}$
$\underline{G}_{BB} \bmod 4 = \{0\}$	$\underline{H}_{BB} \bmod 4 = \{1,2,3\}$

Two trees are generated with roots 0_A and 0_B, using Algorithm 2.9.1. Here we have assumed $i_A < i_B$. The trees are given in Figure 2.10.2. Maximal compatibility classes containing 0_X are:

$$\{0_A, 0_B, 1_B\}, \{0_A, 2_A\}, \{0_B, 1_B, 2_B, 3_B\}, \{0_B, 3_A, 3_B\}. \qquad \square$$

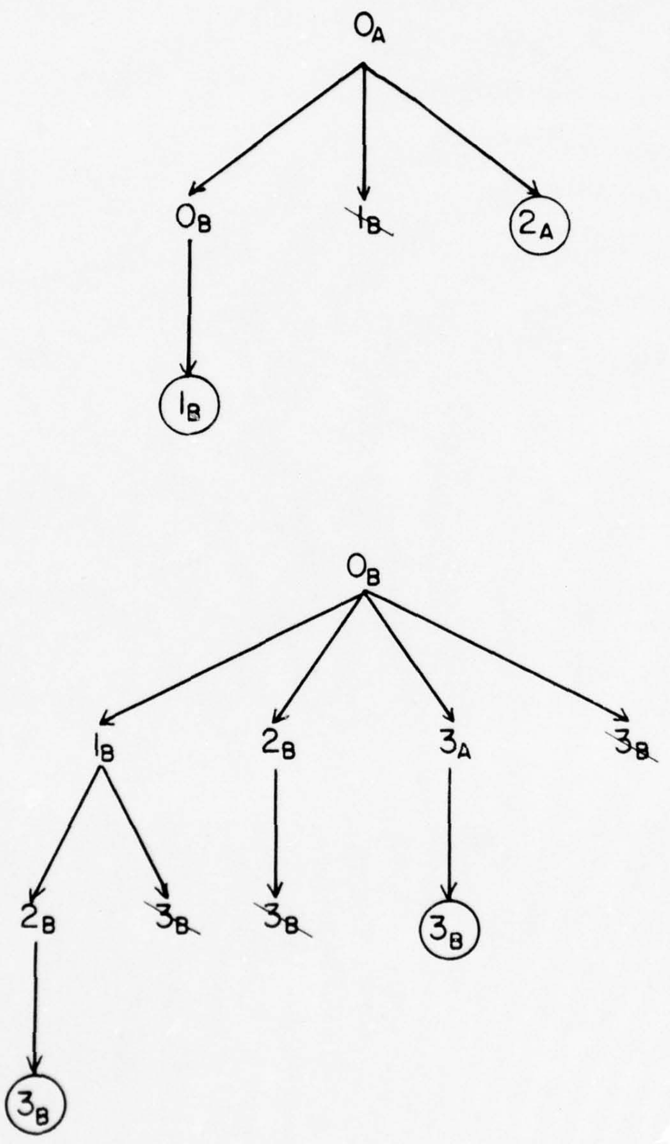
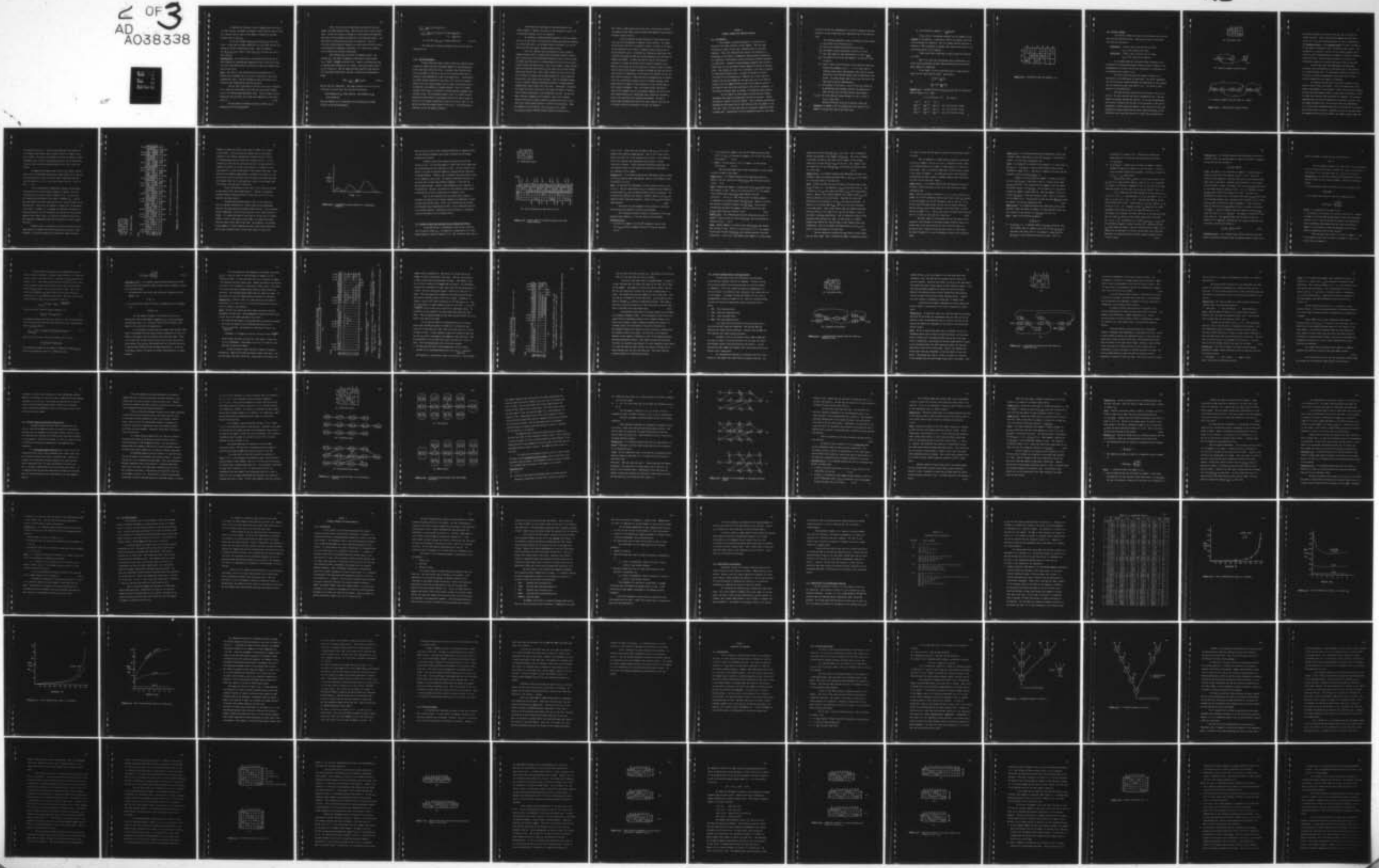


Figure 2.10.2. Generating the maximal compatibility classes of cycle $(1_A, 1_B, 2_A)$.

AD-A038 338 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS.(U)
OCT 76 J H PATEL DAAB07-72-C-0259

UNCLASSIFIED R-747 NL

2 OF 3
AD
A038338



A compatibility class $\underline{C}_1$ is said to <u>cover</u> another class $\underline{C}_2$ if for each function, the number of elements of that function type in class $\underline{C}_1$ is greater than or equal to the number of elements of the same function type in class $\underline{C}_2$.

In the above example $\{0_A, 0_B, 1_B\}$ and $\{0_B, 3_A, 3_B\}$ cover each other. We may apply the same definition for cover among rows and also between a row and a compatibility class. Thus for example a row $\{0_A, 2_A, 3_B, 4_C, 5_B\}$ covers a class $\{0_B, 2_A, 5_A\}$, but does not cover a class $\{0_C, 4_C\}$.

<u>Theorem 2.10.3</u>: For a given cycle, a multifunction pipeline can be made allowable by delaying some computation steps if and only if each row of the reservation table is covered by at least one compatibility class of the cycle.

<u>Proof</u>: The 'only if' part follows directly from Lemma 2.10.3.1, because every allowable row of a cycle is covered by at least some compatibility class. Thus no allowable row can exist which is not covered by some compatibility class of the cycle.

Now the rest of the proof is similar to the proof of Theorem 2.9.5. Each row can be made to look like one of the rows of Lemma 2.10.3.1 by delaying some computation steps. This can always be done because for each row there is at least one compatibility class which covers it. Q.E.D.

Now the problem of making a pipeline allowable can be formulated in the following manner.

First of all we need to generalize the objective function, namely, the added execution delay. Each function has its own execution delay. We may not be able to minimize all these delays simultaneously. Therefore we shall minimize some function of the execution delays. Let D(X) denote the added execution delay for function X. Then the objective function might be thought of as some function of D(X)'s which is monotonically increasing in each D(X); e.g., some linear combination of D(X)'s with positive coefficients.

Each function has its own set of elemental delays. The variable d_{ij}, the number of elemental input delays inserted before row i column j (defined in Section 2.6), needs a third subscript here to indicate which function it is to delay. Thus $d_{ij}(X)$ shall refer to d_{ij} for function X. Now the added execution delay for each function in a reservation table with I rows and J columns can be expressed as below:

$$D(X) = \sum_{0 \leq j < J} \max_{0 \leq i < I} d_{ij}(X). \qquad (2.10.1)$$

Next we form the constraints. The usage interval due to an X in cell (a,b) and a Y in cell (a,c) for $c \geq b$ can be written as

[(the distance of Y_{ac} from column 0) - (the distance of X_{ab} from column 0)].

Thus the complete set of constraints can be expressed from Lemma 2.10.2.2 in the following manner.

$$[(\sum_{0\le j<c} \max_{0\le i<I} d_{ij}(Y) + d_{ac}(Y) + c)$$
$$- (\sum_{0\le j<b} \max_{0\le i<I} d_{ij}(X) + d_{ab}(X) + b)] \bmod p \notin \underline{G}_{XY} \bmod p$$
$$(\text{or } \in \underline{H}_{XY} \bmod p)$$

for each pair $\langle X_{ab}, Y_{ac}\rangle$, ordered arbitrarily. (2.10.2)

The algorithm to obtain an optimum solution is the same as Algorithm 2.7.1.

2.11. Concluding Remarks

We have shown that under certain conditions a pipeline can be modified by simply inserting noncompute segments so that the pipeline allows some given initiation cycle. In general, the condition which must be satisfied is that every row of the reservation table must be covered by some compatibility class of the given cycle. In particular, for single function pipelines we saw that if the cycle is a constant latency cycle with latency greater than or equal to the lower bound of the pipeline, the condition is always satisfied. Thus it is always possible to add delay to a pipeline to achieve maximum throughput with constant cycles. For nonconstant latency cycles in a single function pipeline the lower bound of the pipeline must be less than or equal to the size of the largest compatibility class of the cycle, which in turn may be less than the average latency of the cycle. For multifunction pipelines, we use the general condition.

We presented the techniques for maximum sharing of non-compute segments. However, the effect of such sharing on overall cost is technology dependent and is not presented.

We presented a branch-and-bound algorithm to find an optimum solution (if one exists) to the problem of making the pipeline allowable for a particular cycle. An optimum solution is defined to be the one which uses the least amount of added execution delay.

We do not, however, have an algorithm to find a globally optimum solution; that is, simply given an average initiation latency (and a function mix for multifunction pipelines) find a cycle for which the added execution delay is a minimum among all cycles with the given average latency and function mix. This is by no means a simple problem. There are infinitely many distinct cycles which have the same average latency and function mix. Only a small fraction of these cycles (but still infinite in number) can be made allowable for a given pipeline. We know very little about cycle structure. Even to determine whether a cycle is perfect requires cumbersome procedures of forming maximal compatibility classes except for a few cases. Furthermore, in general, the longer the period of the cycle, the larger the number of compatibility classes (especially so for multifunction cycles). We can only make a heuristic suggestion for choosing a 'good' cycle, namely, choose a cycle with as small a period as possible and choose a cycle which has a fairly regular pattern of latencies and functions. Thus, for example, we would choose a cycle $(4_A, 4_B, 4_C)$ over a cycle $(2_A, 5_B, 3_B, 1_C, 10_A, 3_C)$ with same average latency and function mix.

This choice is made because cycles with only a few distinct latencies are likely to have small G sets and hence more freedom in the choice of allowable usage intervals.

Once the pipeline has been modified to allow some desired cycle how do we accommodate a different arrival pattern? For single function units this is not a difficult problem. A buffer at the input of the pipeline is all we need to change any arrival pattern into a desired allowable cycle. Guidelines for choosing an appropriate size of buffer for random arrivals and constant departures are discussed in [DOR67] and [PHI70]. A buffer can also be used for multifunction pipelines to convert the arrivals into a desired initiation cycle. However, this scheme may be too restrictive, because we have to assume that the function mix remains constant on average and that the order of task arrivals is unimportant, so the tasks can be shuffled to suit the cycle before initiation. These restrictive assumptions can be relaxed if we use a scheduling strategy, such as the greedy strategy, which adapts itself to changes in input patterns, but may possibly have to incur reduced throughput. Thus, even though we have been able to improve the throughput by use of noncompute segments, we still do not have enough flexibility in initiations when a particular initiation cycle is not required by the operating environment. We hope to achieve high throughput and flexibility, when required, with the use of internal buffers as discussed in the next two chapters.

Chapter 3

INTERNAL BUFFERS WITH PERIODIC ARRIVALS

3.1. Introduction

The allowable initiation sequences of a pipeline are a function of the usage intervals of each segment. Once the usage intervals are fixed, we have only a limited choice of initiation sequences. Since this limitation may restrict the flexibility in scheduling, we may think of changing the usage intervals dynamically, for operating in environments in which restricted initiation sequences are undesirable. Instead of using noncompute segments which provide a fixed delay, we can use buffers between segments to provide a variable delay. Once buffers are used, it is convenient to use them to simplify the scheduling problem as well. We shall use buffers to resolve collisions in the following sense. Whenever two or more tasks are trying to use the same segment at the same time, one of the tasks is allowed to use the segment while the rest of the tasks are stored in the buffer to wait their turn according to some priority mechanism. In this sense the buffers might be thought of as adaptive delay elements.

In this chapter we shall examine various priority schemes. Throughout this chapter we assume that arrivals are periodic. Random arrivals are treated in the next chapter. Unless otherwise specifically mentioned we also assume that no task uses more than one segment in one time unit. This is to say that there is no parallel computation within a single task. Justification for this assumption should be clear later on when we discuss the implementation of priority schemes in practice. As before, we also assume that each computation step is one time unit long.

The following symbolism is used consistently in this chapter.

p = the period of an initiation cycle.

$\underline{N}$ = the set of function types present in the cycle.

n_X = the number of task initiations of type X in a cycle.

n = the total number of task initiations in a cycle = $\sum_{X \in \underline{N}} n_X$.

m_{iX} = the number of time units that segment i is used for a task of type X.

T_i = task i, which is the i^{th} arrival at the pipeline, where the arrivals are numbered 0,1,2,... If 2 or more tasks of different types arrive simultaneously then they are ordered arbitrarily, this ordering however, must be maintained every period. We assume though, that no more than one task of the same type can arrive simultaneously.

t_i = the instant in time after t_i units of time have passed by. We assume that any transfers, arrivals and departures of tasks can only occur at these discrete time instants.

$\langle t_i,t_j \rangle$ = the time interval between instants t_i and t_j, which is (t_j-t_i) units long.

Given an initiation cycle and a pipeline, define the <u>workload</u> of a segment to be the average busy time required of the segment to process the tasks at the input rate:

w_i = the workload of segment i = $(\sum_{X \in \underline{N}} n_X m_{iX})/p$.

We may refer to the maximum workload over all segments as the <u>workload</u> of the pipeline. A workload of 100% is the highest load which can be processed by a pipeline. Thus in a single function pipeline, a workload of 100% corresponds to average input latency being equal to the lower bound latency of the pipeline.

Unless specifically mentioned, we shall assume that workload is $\leq$ 100%.

When it is clear that the pipeline under consideration is a single function pipeline, we shall omit the subscript X in all of the above symbols for simplicity.

If the range of X is not specified then the range should be taken over all valid function types. Specifically,

$$\sum_X f(X) \equiv \sum_{X \in \underline{N}} f(X),$$

and

$$\max_X f(X) \equiv \max_{X \in \underline{N}} f(X).$$

<u>Example 3.1.1</u>: Consider the cycle $(2_A, 3_B, 0_A, 2_C, 5_B)$ and the reservation table of Figure 3.1.1. Then

$$p = 2+3+0+2+5 = 12 \qquad \underline{N} = \{A,B,C\}.$$

$n_A = 2$ $\quad n_B = 2$ $\quad n_C = 1$

$m_{0A} = 3$ $\quad m_{0B} = 0$ $\quad m_{0C} = 2$ $\quad w_0 = (6+0+2)/12 = 66.6\%$

$m_{1A} = 2$ $\quad m_{1B} = 3$ $\quad m_{1C} = 1$ $\quad w_1 = (4+6+1)/12 = 91.6\%$

$m_{2A} = 1$ $\quad m_{2B} = 1$ $\quad m_{2C} = 2$ $\quad w_2 = (2+2+2)/12 = 50.0\%$

□

	0	1	2	3	4	5
S_0	A	C		A	C	A
S_1		AB	B	BC	A	
S_2	BC		AC			

Figure 3.1.1. Reservation table for Example 3.1.1.

3.2. Priority Schemes

The priority schemes that come to mind initially are First-In-First-Out and Last-In-First-Out. Therefore we start our discussion with these two priorities.

FIFO-global: A task T_i has a priority over all tasks, T_{i+k} $(k > 0)$, which arrive later.

LIFO-global: A task T_i has a priority over all tasks, T_{i-k} $(k > 0)$, which arrive earlier.

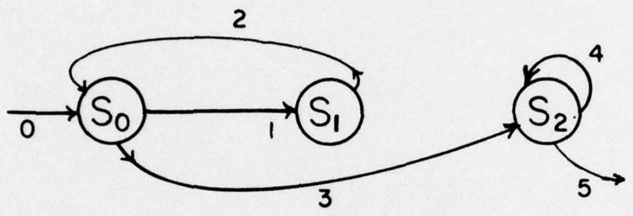
As an example take the reservation table of Figure 3.2.1a. A schematic of this pipeline without buffers is given in Figure 3.2.1b. Implementation of buffers with priority FIFO-global or LIFO-global can be represented as in Figure 3.2.1c.

We should stress here that the buffers in Figure 3.2.1c themselves are not FIFO queues or LIFO stacks. The actual implementation of the priority scheme FIFO-global and LIFO-global is more complex than the impression we get from Figure 3.2.1c. One possible implementation is described below.

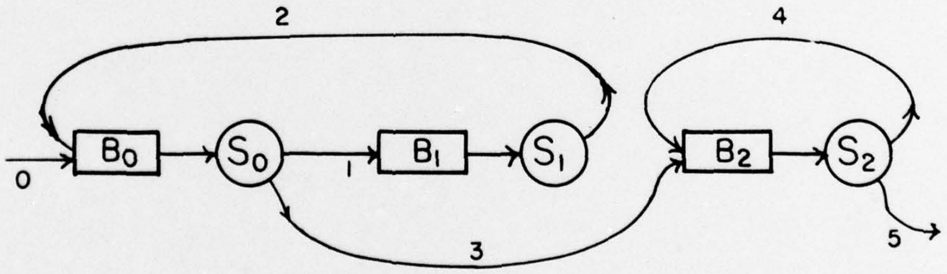
Each task carries a tag which is initialized to zero at its arrival. At every time unit the tag is incremented by one. The tag number now uniquely defines the priority. In the case of FIFO-global, the priority increases with the value of the tag while in the case of LIFO-global a higher valued tag implies a lower priority for that task. Assuming the buffers always have the tasks ordered according to their priorities, every task that arrives at a buffer must be merged into

	0	1	2	3	4
S_0	X		X		
S_1		X			
S_2				X	X

(a) Reservation table.



(b) Pipeline schematic without buffers.



(c) Pipeline schematic with one buffer per segment.

<u>Figure 3.2.1</u>. A pipeline with internal buffers.

the buffer according to the value of its tag, this by itself is a complex operation. When a task uses some segment more than once, it must also carry some information about its state of computation. The computation state or the processing state of a task is defined to be the number of computation steps the task has gone through. This information is required to control the flow between segments and to allow the actual processing done by a segment to differ for different computation steps. One more tag is required for each task in a multifunction pipeline to identify the function type of each task.

We shall show later that FIFO-global and LIFO-global strategies can be implemented in a much simpler way if the pipeline is a single function unit. For multifunction pipelines, however, we do not have a simpler implementation, and we must look for other priority schemes which are simpler to implement.

To visualize the task flow it is useful to think of three distinct steps at each time instant. First, the tasks arrive in the buffers from segments or from outside. Second, the priority is computed and tasks with the highest priority in each buffer are selected. Third, the tasks thus selected leave the buffers for segments. For practical implementation, the priority computation can be done in parallel with the segment execution step, since the task flows are deterministic (i.e., the priority computation has access to the tasks in the buffer as well as those about to arrive). Therefore, as soon as the tasks leave the segments and arrive at the buffers, the highest priority tasks can be selected and sent out. Thus the only additional time required is the transfer time, namely, time to place a task in or remove a task from a buffer. This time can be added to the time to perform a single computation step. Thus for theoretical purposes, one can assume that the above three steps are being performed in zero time at each time instant.

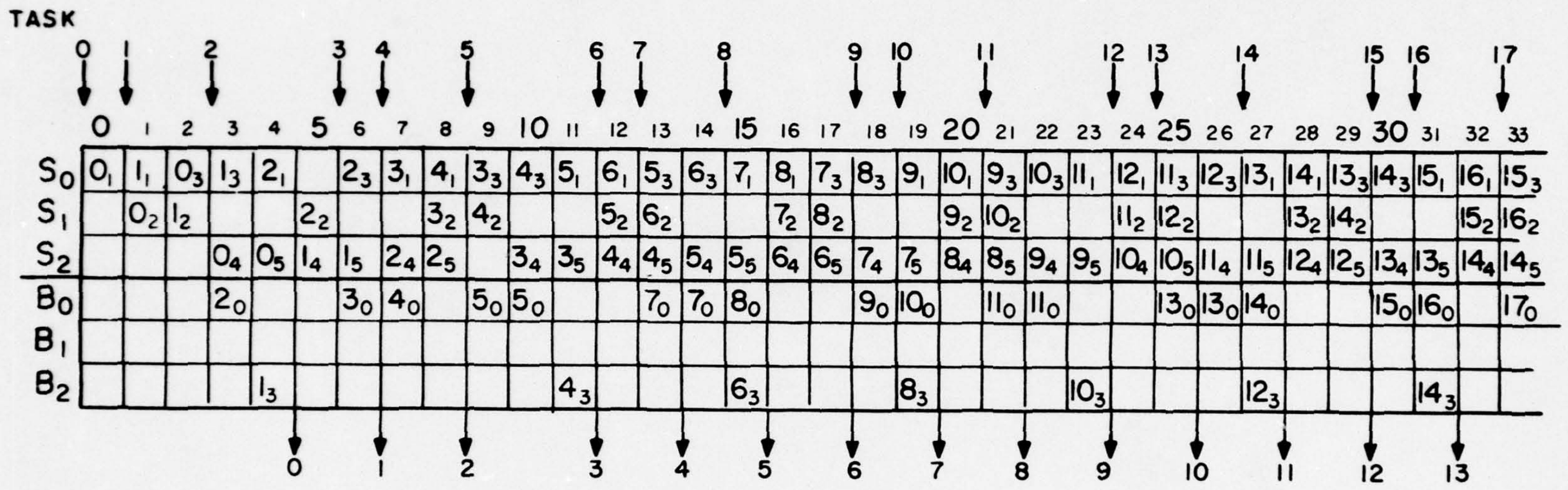
Throughout this chapter when we say "a task leaves a buffer after time t" we mean that task leaves at time instant t+1 or t+2, etc; "a task arrives at a buffer no earlier than t" means that the task does not arrive at t-1 or t-2, etc, but it may arrive at t or t+1 or t+2,.... etc.

As an illustration of FIFO-global strategy, the resulting flow pattern in the single function pipeline of Figure 3.2.2a with initiation cycle (1,2,3) is given in Figure 3.2.2b. B_0, B_1 and B_2 are respectively the buffers at the input of segments S_0, S_1 and S_2. The time instants at which a task arrives at or leaves the pipeline are indicated by ↓. Each unit time interval is marked with the preceding time instant. A number i_j in a segment row indicates that task T_i is being processed for its j^{th} computation step during the time interval marked on the column. And i_j in a buffer row indicates that task T_i has completed j computation steps and is waiting for $(j+1)^{th}$ computation step.

Looking closely at the Figure 3.2.2b one can notice that the usage pattern of segments and buffers becomes periodic after a small transient. To see this consider the columns with labels 10 and 22.

	0	1	2	3	4
S_0	X		X		
S_1		X			
S_2				X	X

(a) Reservation table.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
S_0	0_1	1_1	0_3	1_3	2_1		2_3	3_1	4_1	3_3	4_3	5_1	6_1	5_3	6_3	7_1	8_1	7_3	8_3	9_1	10_1	9_3	10_3	11_1	12_1	11_3	12_3	13_1	14_1	13_3	14_3	15_1	16_1	15_3
S_1		0_2	1_2			2_2			3_2	4_2			5_2	6_2			7_2	8_2			9_2	10_2			11_2	12_2			13_2	14_2			15_2	16_2
S_2				0_4	0_5	1_4	1_5	2_4	2_5		3_4	3_5	4_4	4_5	5_4	5_5	6_4	6_5	7_4	7_5	8_4	8_5	9_4	9_5	10_4	10_5	11_4	11_5	12_4	12_5	13_4	13_5	14_4	14_5
B_0				2_0			3_0	4_0		5_0	5_0			7_0	7_0	8_0			9_0	10_0		11_0	11_0			13_0	13_0	14_0			15_0	16_0		17_0
B_1																																		
B_2					1_3							4_3				6_3				8_3				10_3				12_3				14_3		

(b) Flow in pipeline (a) with cycle (1,2,3).

<u>Figure 3.2.2</u>. *Flow of tasks in a buffered pipeline with FIFO-global priority.*

Suppose we relabel the index of each task by (index - 6) in column 22 and onwards. Since the priority depends only on the relative magnitude of the indices, subtracting a constant from all indices of competing tasks does not affect the original priority. If this relabeling is done then one can easily see that the usage pattern of segments and buffers in column 22 and beyond it is identical to the usage pattern in column 10 and beyond in the original flow chart. This means that the pipeline usage pattern is periodic with period 12. One can also see from the flow chart that the buffers are bounded and that the average output rate of the pipeline is equal to the average input rate, namely; one task every 2 time units.

Recall from Section 2.9 that cycle (1,2,3) cannot be allowed by any pipeline with lower bound latency 2. However, by employing internal buffers we have been able to utilize the cycle (1,2,3) on a pipeline with lower bound 2.

Provided that workload remains $\leq$ 100% one can conjecture that for periodic initiations in a finite pipeline with FIFO-global priority, the usage pattern eventually becomes periodic. This conjecture follows immediately if one can prove that the queues remain bounded in length. If a queue never goes to zero then we can show that it remains bounded. Since; if the queue is never empty, the segment associated with that queue is busy 100% of the time. By assumption, the workload on any segment is $\leq$ 100%, therefore the queue cannot grow indefinitely and it must eventually reach a finite upper bound in finite time.

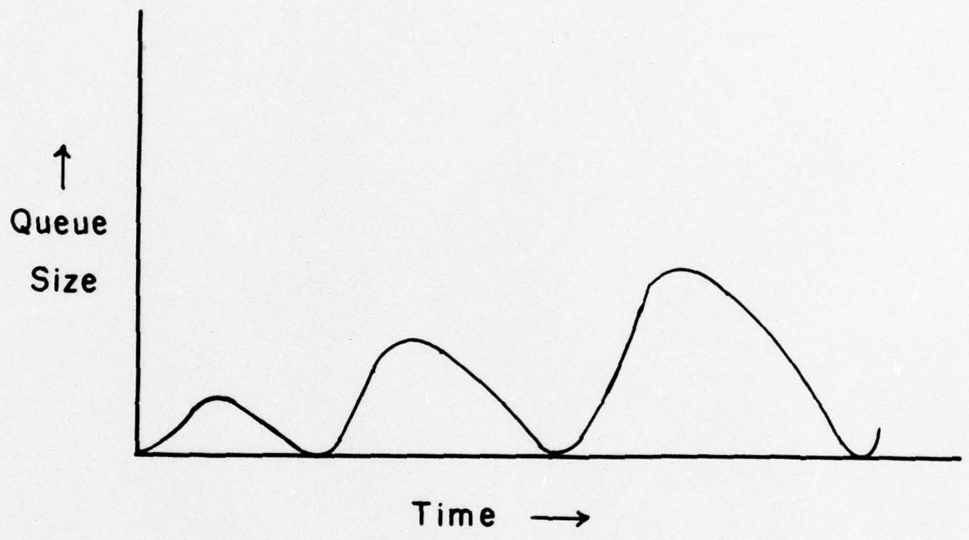


Figure 3.2.3. A hypothetical queue behavior in a FIFO-global pipeline.

However this and similar other reasoning employing the segment utiliza-tion and workload arguments have failed to disprove the following hypothetical situation.

Consider a queue which reaches some peak value and then returns to zero. It then grows again to another peak value higher than the previous peak and then comes back to zero. If this continues (see Figure 3.2.3) then no periodic pattern is achieved and the queue does not remain bounded. However, such a situation has never arisen in any of the examples we have tried. For specific pipelines and cycles we have been able to apply some detailed arguments to show that the queues remain bounded. However, these arguments are too specific to be generalized. Moreover, the details of the arguments almost amount to writing out the flow of tasks such as in Figure 3.2.2b.

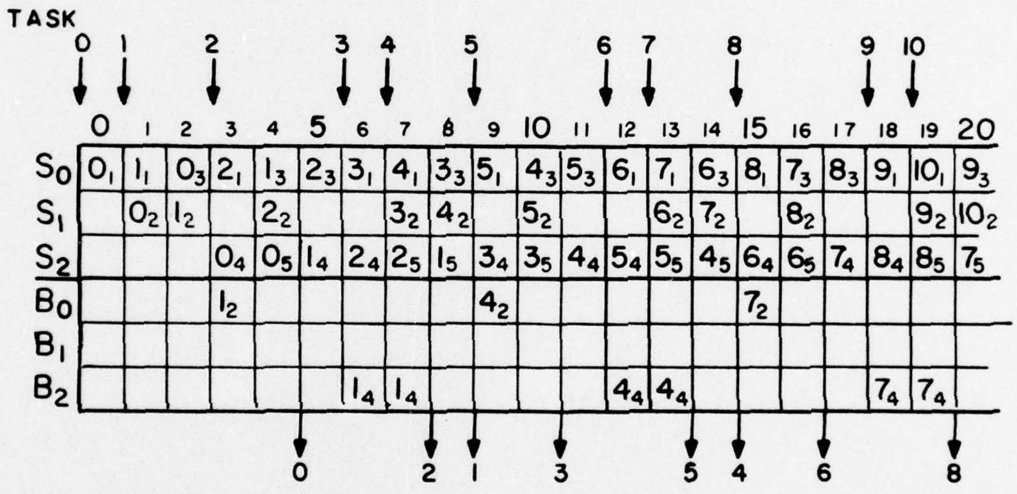
Even if the conjecture that the queues remain bounded is accepted, the question still remains regarding the magnitude of bounds on queue size and wait time. As yet, our intuition has not yielded any reasonable guess regarding the bounds. Thus, FIFO-global priority still remains unsolved. LIFO-global priority,however, is completely solved as discussed in the following section.

3.3. Bounds on Queue Size and Wait Time in LIFO-global Priority

As defined earlier in LIFO-global a task T_i has a priority over all earlier tasks T_{i-k}. An example of a usage pattern in a LIFO-global pipeline is given in Figure 3.3.1b. The initiation cycle used is

	0	1	2	3	4
S_0	X		X		
S_1		X			
S_2				X	X

(a) Reservation table.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S_0	0_1	1_1	0_3	2_1	1_3	2_3	3_1	4_1	3_3	5_1	4_3	5_3	6_1	7_1	6_3	8_1	7_3	8_3	9_1	10_1	9_3
S_1		0_2	1_2		2_2			3_2	4_2		5_2			6_2	7_2		8_2			9_2	10_2
S_2				0_4	0_5	1_4	2_4	2_5	1_5	3_4	3_5	4_4	5_4	5_5	4_5	6_4	6_5	7_4	8_4	8_5	7_5
B_0				1_2						4_2						7_2					
B_1																					
B_2							1_4	1_4					4_4	4_4					7_4	7_4	

(b) Flow in pipeline (a) with cycle (1,2,3).

Figure 3.3.1. Flow of tasks in a buffered pipeline with LIFO-global priority.

cycle (1,2,3). Notice that any two tasks T_i and T_{i+n} ($n = 3$ in this example) have identical usage patterns. This is not a sheer coincidence as we shall see in the theorem soon to follow. It is precisely due to this property that LIFO-global lends itself to a clean theoretical treatment. Unless otherwise specifically mentioned, the word pipeline refers to both single and multifunction pipelines in the remainder of this chapter.

Theorem 3.3.1: In a buffered pipeline with LIFO-global priority, tasks T_i and T_{i+n} have the same usage pattern, where n is the number of task initiations in a cycle.

Proof: By definition of LIFO-global, T_i has a priority over T_{i-k} for all $k > 0$. Thus the usage pattern of T_i is unaffected by any previously initiated tasks T_{i-k} and is governed solely by future initiations. The initiations are periodic and infinite, therefore tasks T_i and T_{i+n} see identical future initiation sequences. Thus T_i and T_{i+n} have identical usage patterns. Q.E.D.

We did not use our assumption on the workload in the above proof and thus the theorem is valid for any workload.

The following corollary is merely a consequence of two usage patterns being identical. It is also valid for any workload.

Corollary 3.3.1.1:

1. T_i arrives at segment s for its k^{th} computation step at time $t \Rightarrow T_{i+jn}$ arrives at segment s for its k^{th} step at time $t+jp$. $\forall j \geq 0$.

2. T_i is processed by segment s for its k^{th} computation step during $\langle t,t+1\rangle \Rightarrow T_{i+jn}$ is processed by segment s for its k^{th} step during $\langle t+jp,t+jp+1\rangle$. $\forall j \geq 0$

3. Segment s is busy during $\langle t,t+1\rangle \Rightarrow$ segment s is busy during $\langle t+jp,t+jp+1\rangle$. $\forall j \geq 0$. □

The following lemmas lead to the establishment of queue bounds as well as bounds on wait times.

Lemma 3.3.2.1: In a buffered pipeline with LIFO-global priority, a segment i does not receive more than $\sum_X m_{iX}n_X$ tasks during any interval $\langle t,t+p\rangle$, $t \geq 0$.

Proof: Suppose some segment i receives more than $\sum_X m_{iX}n_X$ tasks during some interval $\langle t,t+p\rangle$. Then by (1) of Corollary 3.3.1.1, the segment i also receives more than $\sum_X m_{iX}n_X$ tasks during $\langle t+p,t+2p\rangle$, $\langle t+2p,t+3p\rangle$.... and so on. This implies that on average, segment i is subjected to a workload greater than $\frac{1}{p}\sum_X m_{iX}n_X$. However, the input rate implies a workload no greater than $\frac{1}{p}\sum_X m_{iX}n_X$. Thus we have a contradiction and hence the segment i does not receive more than $\sum_X m_{iX}n_X$ tasks during any interval $\langle t,t+p\rangle$. Q.E.D.

Lemma 3.3.2.2: In a buffered pipeline with LIFO-global priority, a segment i cannot be busy for more than $\sum_X m_{iX}n_X$ time units during any interval $\langle t,t+p\rangle$ $t \geq 0$.

Proof: Suppose segment i is busy for more than $\sum_X m_{iX}n_X$ units during some interval $\langle t,t+p\rangle$. Then by (3) of Corollary 3.3.1.1, the segment i is also busy for more than $\sum_X m_{iX}n_X$ units during intervals $\langle t+p,t+2p\rangle$, $\langle t+2p,t+3p\rangle$.... and so on. This implies that segment i, on the average, remains busy for more than $\frac{1}{p} \sum_X m_{iX} n_X$ of the time. This is impossible because the workload of the segment is $\frac{1}{p} \sum_X m_{iX} n_X$. Thus even if segment i processes the tasks at the input rate it cannot, on the average, remain busy for more than $\frac{1}{p} \sum m_{iX} n_X$ of the time. Thus segment i does not remain busy for more than $\sum_X m_{iX} n_X$ units during any interval which is p units long. Q.E.D.

Lemma 3.3.2.3: In a buffered pipeline with LIFO-global priority, and workload $\leq 100\%$, two equally processed tasks T_i and T_{i+jn} $(j > 0)$ cannot be in the same buffer at the same time.

Proof: Suppose a buffer does have two equally processed tasks T_i and T_{i+jn} for some $j \geq 1$. By definition, T_{i+jn} has priority over T_i. Thus if either T_i or T_{i+jn} ever leave the buffer, T_{i+jn} would leave earlier than T_i. However, this violates Theorem 3.3.1, namely, that T_{i+jn} must follow T_i exactly jp time units later. Therefore the only other possibility is that neither T_i nor T_{i+jn} ever leaves the buffer. By Theorem 3.3.1 all such tasks T_{i+kn} (for all $k \geq 0$) would accumulate in the same buffer. This in turn implies that the segment associated with this buffer is busy 100% of the time and the number of tasks in the buffer is growing indefinitely. This is impossible because of the assumption that workload is $\leq 100\%$. In other words if a segment remained busy 100% of the time the queue at that segment cannot grow indefinitely. Thus no two equally processed tasks T_i and T_{i+jn} can reside in the same buffer at the same time. Q.E.D.

One can immediately establish an upper bound on queue length from the above lemma. This is because the number of computation states of a task is finite and the number of distinct task-subscripts modulo n is finite.

Thus for example in a single function pipeline a task which arrives at a segment i can be in one of m_i computation states, where m_i as defined earlier is the number of times segment i is used for a single task. Thus, by the above lemma there can be at most $m_i n$ tasks in the buffer at segment i. By taking into account some additional considerations we shall present a somewhat tighter upper bound later, but first we prove some more lemmas.

Lemma 3.3.2.4: In a buffered pipeline with LIFO-global priority and workload $\leq$ 100%, a task waits at most (p-1) time units for a single computation step.

Proof: From the corollary to Theorem 3.3.1, if a task T_i arrives at a segment for its k^{th} computation step at time t then a task T_{i+n} arrives at the same segment for its k^{th} computation step at time t+p. If T_i did not leave the buffer before t+p then the two equally processed tasks T_i and T_{i+n} would reside together in the same buffer, a contradiction to Lemma 3.3.2.3. Thus T_i must leave the buffer before time t+p, that is, it waits no more than (p-1) time units. Q.E.D.

The above lemma is our assurance that all the tasks will be processed. This assurance is necessary because intuition leads us to believe that in the last-in-first-out type of priority there is a possibility that a particular task may wait forever in some buffer. The bound of Lemma 3.3.2.4 can be tightened as shown in the following lemma.

<u>Lemma 3.3.2.5</u>: In a buffered pipeline with LIFO-global priority and workload $\leq$ 100% a task waits no more than $(\sum_X m_{iX}n_X) - 1$ time units at segment i for any single computation step.

<u>Proof</u>: By assumption the workload of any segment i is $\leq$ 100%, that is, $\sum_X m_{iX}n_X \leq p$. If $\sum_X m_{iX}n_X = p$ then we have already proved the above statement in Lemma 3.3.2.4. Therefore it remains to be shown that the statement is also true when $\sum_X m_{iX}n_X < p$.

Suppose a task T_k arrives at segment i at time t and waits for more than $\sum_X m_{iX}n_X - 1$ time units, that is, it waits for at least $\sum_X m_{iX}n_X$ time units. This implies that the segment i was busy during the entire interval $\langle t, t + \sum_X m_{iX}n_X \rangle$. Now by Lemma 3.3.2.4, T_k must be accepted by segment i before time t+p. Thus the segment is busy for at least $(\sum_X m_{iX}n_X) + 1$ time units during $\langle t, t+p \rangle$. This is a contradiction to Lemma 3.3.2.2. Thus T_k does not wait more than $(\sum_X m_{iX}n_X)-1$ units at segment i for any one computation step. Q.E.D.

<u>Lemma 3.3.2.6</u>: In a buffered pipeline with LIFO-global priority and workload $\leq$ 100%, the buffer at segment i is empty during at least one unit interval in the interval $\langle t, t + \sum_X m_{iX}n_X \rangle$ for all $t \geq 0$.

<u>Proof</u>: Again by the workload assumption,

$$\sum_X m_{iX}n_X \leq p.$$

(1) Let $\sum_X m_{iX}n_X < p$. A nonempty buffer for $\sum_X m_{iX}n_X$ successive time units implies that its segment is busy for at least $(\sum_X m_{iX}n_X) + 1$ successive time units, that is, the segment i remains busy for $(\sum_X m_{iX}n_X) + 1$ units during some interval $\langle t, t+p \rangle$. This is a

contradiction to Lemma 3.3.2.2. Therefore the buffer must remain empty for at least one time unit during any interval $\langle t, t + \sum_X m_{iX} n_X \rangle$.

(2) Let $\sum_X m_{iX} n_X = p$. Suppose that the buffer at segment i remains nonempty throughout the interval $\langle t, t+p \rangle$ for some t. Let the p tasks processed successively by segment i during that interval be $T_{i_1}, T_{i_2}, \ldots, T_{i_p}$ respectively.

Consider some task T_{j_1} which is in the buffer during $\langle t, t+1 \rangle$. T_{j_1} must be of lower priority than T_{i_1}, that is, $j_1 < i_1$. Now either T_{j_1} remains in the buffer throughout $\langle t, t+p \rangle$ or it is one of the tasks processed during $\langle t+1, t+p \rangle$, that is, $j_1 = i_{k'}$ for some k' = 2,3,...,p.

If T_{j_1} remains in the buffer throughout $\langle t, t+p \rangle$, then $j_1 < i_k$ for all k = 1,2,...,p. Alternatively, if $j_1 = i_{k'}$ for some k' = 2,...,p, then $j_1 < i_k$ for all k = 1,2,...,k'-1. Consider some task T_{j_2} in the buffer during $\langle t+k'-1, t+k' \rangle$. We must have $j_2 < i_{k'} = j_1$, that is, $j_2 < i_k$ for all k = 1,2,...,k'. Similarly, either T_{j_2} remains in the buffer throughout $\langle t+k'-1, t+p \rangle$ or $j_2 = k''$ for some $k'' = k'+1, k'+2, \ldots, p$. Continuing in this fashion, by finite induction, we can show that there must be a task T_{j_k} which is in the buffer during $\langle t+p-1, t+p \rangle$ with $j_k < i_1, i_2, \ldots, i_p$. By Theorem 3.3.1, tasks $T_{i_1+n}, T_{i_2+n}, \ldots, T_{i_p+n}$ will be processed by segment i during $\langle t+p, t+2p \rangle$ etc., implying that task T_{j_k} never leaves the buffer. This is a contradiction to Lemma 3.3.2.4, namely that no task waits in a buffer for more than (p-1) time units. Therefore, the supposition that the buffer remains nonempty during $\langle t, t+p \rangle$ is false. Q.E.D.

<u>Theorem 3.3.2</u>: In a buffered pipeline with LIFO-global priority and workload $\leq$ 100%, the maximum number of tasks in the buffer at segment i, during any unit time interval is

$$\leq \sum_X m_{iX} n_X - \max_X (n_X).$$

<u>Proof</u>: By Lemma 3.3.2.6, the buffer at segment i is empty during at least one time unit in the interval $\langle t,t+p \rangle$, $\forall t \geq 0$. Since the behavior of the pipeline is periodic with period p, the buffer becomes empty every p time units. Consider a unit interval $\langle t'-1,t' \rangle$ during which the buffer is empty. By Lemma 3.3.2.1 at most $\sum_X m_{iX} n_X$ tasks arrive at the buffer during an interval $\langle t,t+p \rangle$, $\forall t \geq 0$. Therefore, at most $\sum_X m_{iX} n_X$ tasks arrive in the interval $\langle t',t'+p \rangle$. The worst case accumulation of tasks occur in the buffer when these $\sum_X m_{iX} n_X$ tasks arrive at the maximum possible rate. During any unit interval $\langle t,t+1 \rangle$ no more than m_{iX} tasks of type X can arrive at segment i. Thus it takes at least n_X time units for $m_{iX} n_X$ tasks of type X to arrive at the buffer of segment i. Thus the shortest interval in which all these $\sum_X m_{iX} n_X$ tasks can arrive is $\langle t',t'+\max_X (n_X) \rangle$. The segment i processes one task every time unit. Thus $\max_X (n_X)$ tasks leave the buffer during the interval $\langle t',t'+\max_X (n_X) \rangle$, implying that the largest accumulation that could have occurred in the buffer is

$$\sum_X m_{iX} n_X - \max_X (n_X) \text{ tasks.}$$ Q.E.D.

<u>Corollary 3.3.2.1</u>: In a buffered single function pipeline with LIFO-global priority and workload $\leq$ 100%, the maximum number of tasks in the

buffer at segment i, during any unit time interval is

$$\leq m_i n - n. \qquad \square$$

For the example in Figure 3.3.1a and cycle (1,2,3) the maximum sizes of buffers from the above corollary are: $B_0 \leq 3$, $B_1 \leq 0$ and $B_2 \leq 3$. From Figure 3.3.1b, the actual sizes are: $B_0 = 1$, $B_1 = 0$ and $B_2 = 1$.

<u>Theorem 3.3.3</u>: In a buffered pipeline with LIFO-global priority and workload $\leq$ 100%, the wait times at segment i satisfy the following bounds:

1. The maximum wait time of any task for any one computation step

$$\leq (\sum_X m_{iX} n_X) - 1.$$

2. The expected wait time of a task for a computation step

$$\leq \frac{1}{2} \sum_X m_{iX} n_X - \frac{\sum_X m_{iX} n_X^2}{2 \sum_X m_{iX} n_X} .$$

<u>Proof</u>: 1. Already proven in Lemma 3.3.2.5.

2. The behavior of the pipeline is periodic with period p. Moreover by Lemma 3.3.2.6 the buffer becomes empty at least once during each interval $\langle t,t+p \rangle$. Thus we can find the bound on the expected wait time by considering the maximum total wait time of the tasks at segment i, in an interval $\langle t,t+p \rangle$ and averaging over the total number of computation steps in that interval.

The total wait time of tasks in $\langle t,t+p \rangle$ at segment i = (sum of the departure times from the buffer at segment i) - (sum of the arrival times at segment i).

To get the worst case wait time, we assume that arrivals occur as early as possible. Consider a unit interval $\langle t-1,t\rangle$ when the buffer is empty. By Lemma 3.3.2.1, a maximum of $\sum_X m_{iX}n_X$ tasks arrive during $\langle t,t+p\rangle$. The arrivals occur with the restriction that no more than m_{iX} tasks of type X arrive during any one time unit and that a total of $m_{iX}n_X$ tasks of type X arrive during $\langle t,t+p\rangle$. Thus by the worst case assumption m_{iX} tasks of type X arrive every time unit starting at t, for n_X consecutive time units. Thus the sum of arrival times of tasks of type X in the interval $\langle t,t+p\rangle$ is,

$$\sum_{0\leq j<n_X} (t+j)m_{iX} = m_{iX}n_X t + \frac{m_{iX}n_X(n_X-1)}{2} .$$

Then the sum of all arrival times at segment i is,

$$\sum_X[m_{iX}n_X t + \frac{1}{2} m_{iX}n_X(n_X-1)]. \qquad (a)$$

The segment processes one task every time unit. Thus one task departs from the buffer every time unit and therefore the sum of the departure times of $\sum_X m_{iX}n_X$ tasks is

$$\sum_{0\leq j<\sum_X m_{iX}n_X} (t+j) = \sum_X m_{iX}n_X t + \frac{1}{2} (\sum_X m_{iX}n_X)((\sum_X m_{iX}n_X)-1). \qquad (b)$$

Then the sum of the wait times in the worst case is (b)-(a)

$$= \frac{1}{2} (\sum_X m_{iX}n_X)^2 - \frac{1}{2} \sum_X m_{iX}n_X^2.$$

This quantity averaged over the number of computation steps $\sum_X m_{iX}n_X$, gives the expected wait time for a computation step,

$$= \frac{1}{2} \sum_X m_{iX} n_X - \frac{1}{2} \frac{\sum_X m_{iX} n_X^2}{\sum_X m_{iX} n_X} \quad .$$ Q.E.D.

Corollary 3.3.3.1: In a buffered single function pipeline with LIFO-global priority and workload $\leq$ 100%, the wait times at segment i satisfy the following bounds:

1. The maximum wait time of any task for any one computation step at segment i is,

$$\leq m_i n - 1.$$

2. The expected wait time of a task for a computation step at segment i is,

$$\leq \frac{1}{2} (m_i n - n).$$ □

For the example in Figure 3.3.1a and cycle (1,2,3) the maximum wait time in the buffer at segment S_2 is ≤ 5 and the maximum expected wait time $\leq 3/2$. The actual values for this example, from Figure 3.3.1b, are 2 and 1/3 respectively.

In the majority of cases the actual queue sizes and wait times remain considerably below the bounds stated in Theorems 3.2.2 and 3.3.3. However, the bounds, with the exception noted below are tight. Tight in the sense that for each bound on queue size and wait time and any given values of m_{iX} and n_X, there always exist an initiation cycle and a pipeline which actually attain the upper bound values. It may not be possible, however, to achieve all bounds simultaneously in a single pipeline.

The one exception to the tightness of the bounds occurs when $\sum_X m_{iX} = 1$, that is, there is only one usage of segment i in the reservation table. In this case there can be no collisions at segment i and therefore the wait time is zero. However the bound on the maximum wait time in (1) of Theorem 3.3.3 may have a nonzero value. The bounds on the queue size and the expected wait time are tight without any exception. The following theorem shows that the bounds are tight. We have shown this only for single function pipelines for simplicity.

Theorem 3.3.4: Given $m > 1$ and $n > 0$, there exist for each bound of Corollaries 3.3.2.1 and 3.3.3.1, a cycle and a pipeline such that the bound is reached in the pipeline.

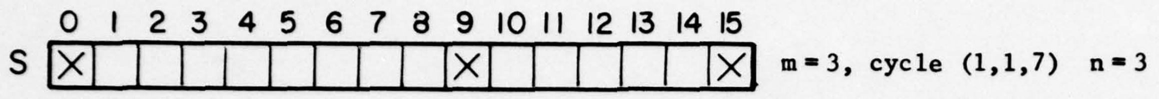
Proof: We shall only sketch the proof rather than give a detailed explanation of every step. The accompanying illustrations should help clarify most of the assertions made in this proof.

1. We claim that the bounds on the maximum queue size and the expected wait time are reached in the following pipeline with cycle $(\underbrace{1,1,\ldots,1}_{(n-1)\ 1\text{'s}},mn-n+1)$. The pipeline is constructed to have a row

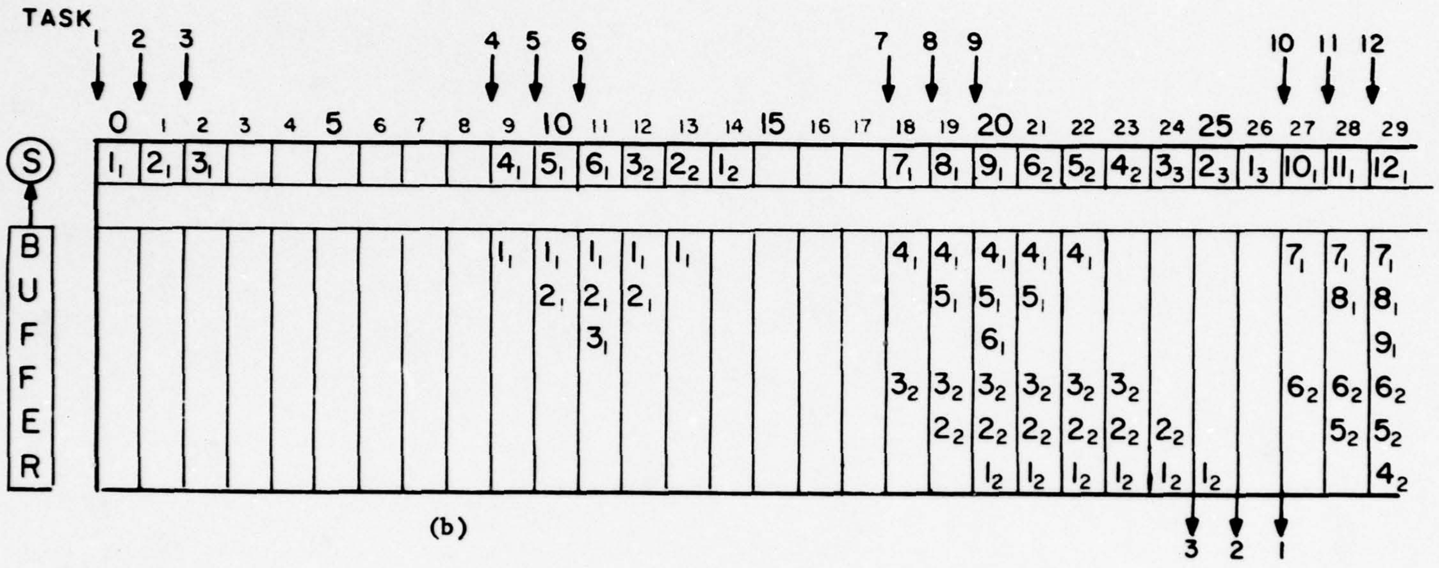
   $$\{j \mid \text{integer } j, \exists \text{ integer } k, 0 \le k \le m-1 \text{ such that } j = kmn - \frac{(k-1)kn}{2}\}$$

   and all other rows with one X per row. The number of these rows is of no consequence. (See Figure 3.3.2a, for $m = 3$ and $n = 3$, the rows with one X are not shown.)

To prove the above claim, note that n tasks are initiated successively. They do not suffer any delay at their first step. (In this discussion the first, second etc. steps refer to the steps of the



(a)



(b)

1. Maximum queue size occurs during interval ⟨20,21⟩ = 6 tasks.
   Bound = mn - n = 3 × 3 - 3 = 6 tasks.

2. Expected wait time of a task at a computation step = $\dfrac{\text{sum of wait times of } T_1, T_2, T_3}{\text{total No. of computation steps}} = \dfrac{11+9+7}{9} = 3.$

   Bound = (mn-n)/2 = 3 time units.

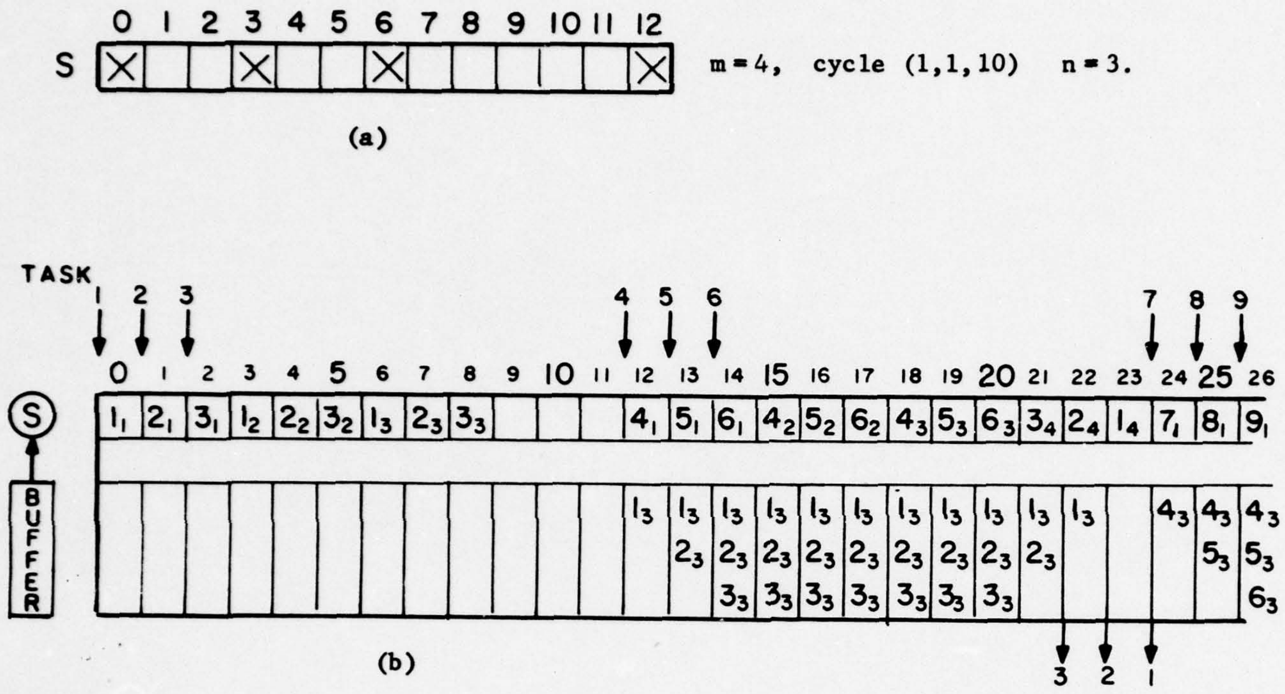
<u>Figure 3.3.2</u>. Example showing tightness of bounds on queue size and expected wait time with LIFO-global priority.

segment under consideration.) The arrival for second step on the segment is exactly one period ($=mn$) later. Thus the first group of n tasks collide with the second group of tasks. (See Figure 3.3.2b for the flow of tasks in the segment and its buffer.) By LIFO-global priority the first group of n tasks has to wait for the second group of n tasks. The first group thus suffers a total of n^2 time units of delay. The first group then collides at the third step with groups 2 and 3 and it suffers a total of $2n^2$ units of delay. Similarly, it suffers a total of $3n^2$ units of delay at the 4th step and so on. Thus the first group of n tasks suffers a total of $0+n^2+2n^2+\dots+(m-1)n^2 = = \frac{m(m-1)n^2}{2}$ units of delay for a total of mn computation steps. By periodicity of the pipeline, all other groups also suffer the same delay. Then the expected delay at each step is $\frac{1}{2}\, m(m-1)n^2/mn = \frac{1}{2}\,(m-1)n$. which is the same as the bound.

To see how the bound on the maximum queue size is reached notice that m different groups of n tasks each, collide with each other over a time interval n units long (e.g. time units 18, 19 and 20 in Figure 3.3.2b). In that interval m tasks are received at each time unit and one task is processed by the segment at each time unit. Thus an accumulation of (m-1) tasks occur every time unit for a total of n units. Thus the maximum queue size of (m-1)n is reached.

2. We claim that the bound on the maximum wait time at a step is reached in the following pipeline with cycle $(\underbrace{1,1,\dots,1}_{(n-1)\ 1\text{'s}},mn-n+1)$.

This pipeline is constructed to have a row $\{0,n,2n,\dots,(m-2)n\}\cup\{mn\}$;



Wait time of task T_1 at the 4th computation step = 11 time units.

Bound on maximum wait time = mn - 1 = 4 X 3 -1 = 11 time units.

Figure 3.3.3. Example showing the tightness of bound on maximum wait time with LIFO-global priority.

and all other rows have one X per row. (See Figure 3.3.3a for $m = 4$ and $n = 3$, the rows with one X are not shown.)

The proof of the above claim is straightforward. A group of n tasks initiated does not suffer any delay for the first (m-1) steps on the segment. (See Figure 3.3.3b for the flow of tasks.) For the m^{th} step the same group collides with the next incoming group of n tasks. The second group uses the segment for (m-1)n time units. Thus the task T_0 is delayed for (m-1)n time units. T_0 also waits for the tasks T_1 through T_{n-1} because of LIFO-global priority. Thus task T_0 waits for a total of $(m-1)n + (n-1) = mn-1$ time units at the m^{th} step. This is the same as the upper bound value. Q.E.D.

We mentioned earlier that the internal buffers can be thought of as adaptive elemental delays. The best example of this occurs in a single function LIFO-global pipeline with a constant latency cycle. Recall the procedure to make a pipeline allowable with respect to a constant latency cycle described in the proof of Theorem 2.3.1. The elemental delays were inserted to avoid collisions, starting from the leftmost column to the right. The task flow in the pipeline modified in this manner is identical to that of the original pipeline employing LIFO-global buffers. This happens because when modifying the table, we inserted delay elements of a later computation step, which is to say that when two tasks collide, the earlier arriving task is always delayed over the later arriving task. This delay insertion procedure amounts to a LIFO-global priority.

3.4. Priority Schemes Based on Processing State

We mentioned earlier that FIFO-global and LIFO-global priorities were somewhat difficult to implement. We shall see that the priorities based on the processing state of a task are simpler to implement. In all such schemes we assume an implementation in which one buffer is provided for each computation step rather than one at each segment. A schematic showing the buffers for this type of implementation is given in Figure 3.4.1b. What the following schemes describe are priorities among buffers rather than those within a buffer.

1. MPF: Most Processed First.
2. LWRF: Least Work Remaining First.
3. LPF: Least Processed First.
4. MWRF: Most Work Remaining First.

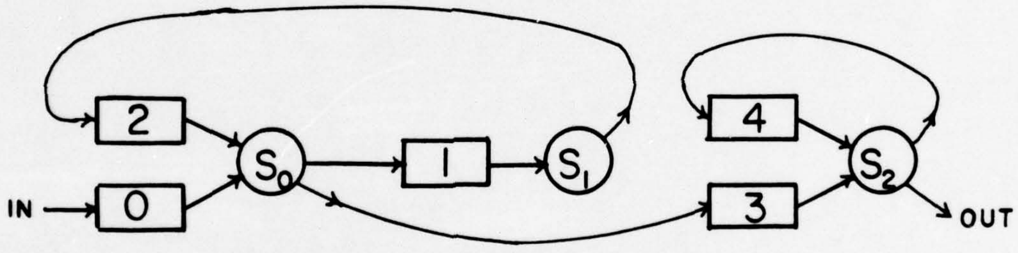
In a single function pipeline the most processed task is also the task with least work remaining. Thus MPF and LWRF are identical for single function pipelines. Similarly LPF and MWRF are also identical for single function pipelines.

In Figure 3.4.1b if the priority used is MPF or LWRF then the tasks in buffer 2 would have priority over the tasks in buffer 0 and similarly buffer 4 would have priority over buffer 3. If the priority was LPF or MWRF then buffers 0 and 3 would have priority over buffers 2 and 4 respectively.

In a multifunction pipeline it is possible that two or more buffers at some segment have tasks which are equally processed. For

	0	1	2	3	4
S_0	X		X		
S_1		X			
S_2				X	X

(a) Reservation table.



(b) Schematic with buffers.

Figure 3.4.1. A single function pipeline with one buffer per computation step.

example buffers 1_A and 1_B in Figure 3.4.2b have tasks which have completed 1 step. Thus MPF and LPF schemes would not define the priority uniquely. Similarly the buffers 2_A and 3_B in the same figure have tasks which have an equal amount of work remaining, namely, 1 step. And therefore LWRF and MWRF do not assign the priority uniquely. Whenever such ambiguity arises we shall assume that some arbitrary decision is made on the priority among competing buffers. However, once such priority is assigned we assume that it remains fixed.

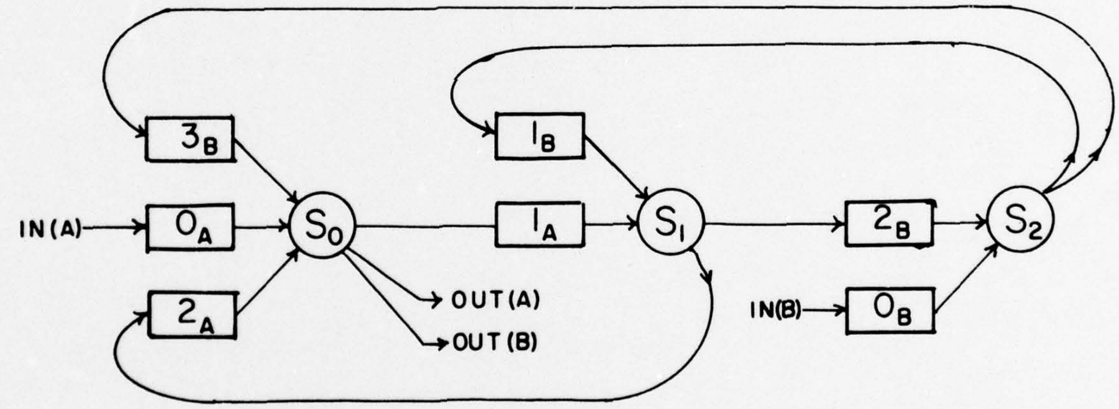
A question still remains regarding the priority among tasks within a buffer. This priority is of little consequence as stated below.

Theorem 3.4.1: In pipelines, which use a priority based on processing state and use one buffer per computation step, the queue size at any instant in time, the average wait time of a task in a buffer and the pipeline throughput are independent of the priority used among tasks within a buffer.

Proof: As far as the current queue size is concerned, it is of no consequence which task is chosen to leave the buffer. All the tasks within a buffer are of the same function type and are in the same state of computation. And because the priority among buffers is based on the processing state, any task leaving the buffer will have the same effect on the flow of other tasks. This in turn implies that future queue sizes are independent of the task chosen to leave the buffer. The average wait time of a task in a buffer is a function of the average queue size and average output rate of the buffer. Both

	0	1	2	3
S_0	A		A	B
S_1		AB		
S_2	B		B	

(a)



(b)

Figure 3.4.2. A multifunction pipeline with one buffer per computation step.

of which are independent of the priority within a buffer. The throughput is a function of average output rate of the buffer. Again both of these are independent of the priority within a buffer. Thus the queue size, the average wait time and the throughput are independent of the priority within a buffer. Q.E.D.

Also note that the above theorem is valid whether the input is periodic or random. We should also stress that it is the average and not necessarily the maximum or variance of the wait time which is independent of the priority within a buffer.

A complete priority description will be denoted by the priority among buffers followed by the priority within the buffer. For example, LPF, FIFO-buffer, or MWRF, LIFO-buffer, etc. In any discussion when the priority within a buffer is unimportant, we shall not specify it but shall simply refer to the priorities among the buffers.

Priorities based on processing states are considerably simpler to implement than FIFO-global or LIFO-global priorities. Since we have assumed one buffer per computation step, it is seen that each buffer has a unique successor buffer and a unique predecessor buffer. Thus the control of the task flow is simple. Moreover, since a buffer holds only tasks of a single function type and a single state of computation, no tag is necessary to identify the state of a task. The priority among buffers is fixed and hence no elaborate control mechanism is required to order the tasks. The priority within a buffer can be chosen so as to make its implementation simple; e.g., FIFO or LIFO priority.

We stated earlier in Section 3.2 that FIFO-global and LIFO-global can sometimes be implemented in a simpler way than the one described. In the following we show that in some cases FIFO-global and LIFO-global are equivalent to some of the priorities described in this section.

<u>Theorem 3.4.2</u>: The flows of tasks in a single function pipeline are identical under the following priorities.

1. FIFO-global. 2. MPF, FIFO-buffer. 3. LWRF, FIFO-buffer.

<u>Proof</u>: MPF and LWRF are identical in a single function pipeline. Therefore the task-flows are identical under 2 and 3. Next we show that the task-flows are also identical under 1 and 2.

Consider any two tasks T_i and T_j $(j > i)$. With FIFO-global priority T_i has a priority over T_j everywhere in the pipeline. With MPF, FIFO-buffer, since all buffers are first-in-first-out and T_i initially arrives before T_j, T_i always is either equally processed or more processed than T_j and T_i arrives before T_j at each step. Thus the task T_i has priority over T_j everywhere in the pipeline. Therefore the two priority schemes result in the same flow of tasks. Q.E.D.

<u>Theorem 3.4.3</u>: The flows of tasks in a single function pipeline employing a constant latency initiation cycle are identical under the following priorities.

1. LIFO-global 2. LPF, *-buffer 3. MWRF, *-buffer

where * stands for any arbitrary priority.

Proof: Let us consider what happens under LIFO-global priority. By Theorem 3.3.1, tasks T_i and T_{i+n} have identical flow patterns under LIFO-global. For a constant latency cycle, n is 1 and therefore every task has the same flow pattern. Thus in LIFO-global for any two tasks T_i and T_j, $j > i$, which are in different states of computation, T_j is less processed than T_i. Suppose these tasks are competing for the same segment, then under LIFO-global T_j has priority over T_i because $j > i$.

By LPF T_j also has priority over T_i, because T_j is less processed than T_i. Thus we can substitute LPF for LIFO-global whenever two or more tasks in different states of computation are competing for a segment.

Now by Lemma 3.3.2.2, under LIFO-global no two equally processed tasks T_i and T_{i+nj}, $j > 0$, can be in the same buffer, and $n = 1$ implies that no two equally processed tasks will ever compete for the same segment. If we provide a separate buffer for each computation step, there will never be more than one task in any buffer and hence no priority is required within a buffer. Thus we have transformed the LIFO-global discipline into a LPF, *-buffer implementation without affecting the task-flow.

In a single function pipeline LPF is identical to MWRF and therefore the task-flow is also the same under MWRF, *-buffer.

Q.E.D.

In the next section we will see that LIFO-global, LPF and MWRF are similar, but not identical, under periodic but nonconstant initiation. We have seen in Theorem 3.4.2 that FIFO-global, MPF and LWRF are also similar, but we are not able to make any concrete comments regarding any one of them. Some bounds are presented for a priority scheme called Precedence-Implied Priority (PIP) in the following section, where LPF and MWRF are shown to be the special cases of that particular priority scheme.

3.5. Bounds on Queue Size and Wait Time with PIP

Precedence-Implied Priority (PIP) is introduced in this section. When LPF and MWRF priorities are applied to multifunction pipelines there is ambiguity in the priority scheme. PIP is a class of priorities which includes all priorities with consistent resolution of this ambiguity. Most of the theorems to follow are stated in terms of PIP and are thus directly applicable to LPF, MWRF and their extensions.

In a Precedence-Implied Priority (PIP) scheme, for any two computation steps x and y, step x precedes step y implies that step x of one task has priority over step y of another task. Moreover, we consider only "consistent" priority schemes in the sense that they must be transitive, that is, step x has priority over step y and step y over step z implies that step x has priority over step z. It is understood that any priority scheme must be complete in the sense that the priority between any two steps that use the same segment is defined.

Note that sometimes two steps performed on two distinct segments may have a priority relation by the above definition, even though these two steps can never collide with each other. However, the priority between such steps should not be ignored since it implies other meaningful priorities through transitivity.

Note also that the precedence relation is not always sufficient to define the priority among all steps, since there does not exist any precedence relation between two steps of two distinct function types. Thus one still has considerable freedom in assigning the priorities required for completeness after the precedence implied priorities have been assigned. Therefore, PIP actually represents a class of priorities.

In a single function pipeline all the steps are related by precedence and therefore PIP defines a unique priority, which is clearly identical to LPF and MWRF priorities. In a multifunction pipeline PIP still assigns a LPF priority on each individual function but leaves the priority among steps of distinct functions unassigned.

To determine whether a priority scheme is of type PIP we can look at the graph of the precedence and priority relation. The graph consists of a set of nodes and directed edges. Each node represents a distinct computation step in the pipeline. A directed edge exists from a node x to node y iff either step x precedes step y or step x has a priority over step y. The priority is of type PIP if and only if the graph is acyclic (i.e., does not have a closed directed path). A PIP graph is acyclic since the relation on the graph, namely, x related to y ($x < y$) iff x precedes y or x has a priority over y is transitive ($x < y$, $y < z \Rightarrow x < z$), irreflexive ($x \nless x$) and hence assymetric ($x < y \Rightarrow y \nless x$). This relation resembles a partial ordering and many well-known authors have loosely referred to it as a partial ordering (e.g. [KNU73] pp. 258-259). For details on relations and their graphs see any book on modern algebra (e.g. [STO73]). For readability, edges may be deleted from a precedence-priority graph if they would be implied by transitivity.

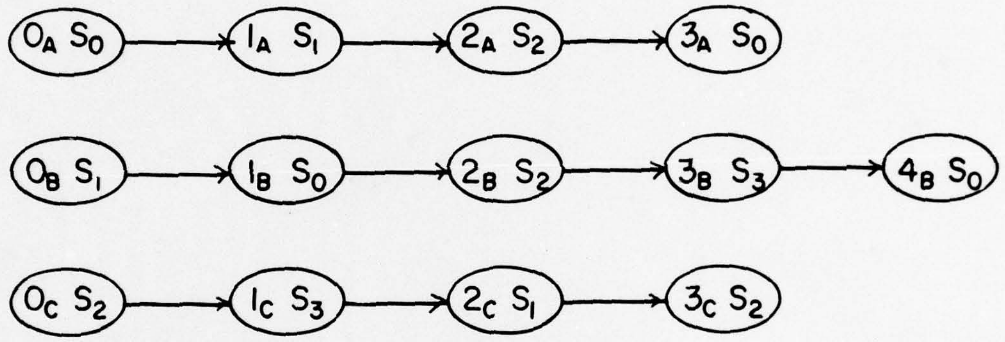
As an example, take the pipeline of Figure 3.5.1a. Figure 3.5.1b is the precedence graph of that pipeline. The nodes in the graph are labeled with the number and type of the computation step as well as the name of the segment on which the step is performed. As an example, a precedence-priority graph of a priority of type PIP on the same pipeline is given in Figure 3.5.1c.

We already mentioned the similarity of LPF and MWRF priorities to PIP. With the help of precedence-priority graphs we show below that LPF and MWRF priorities are indeed of type PIP.

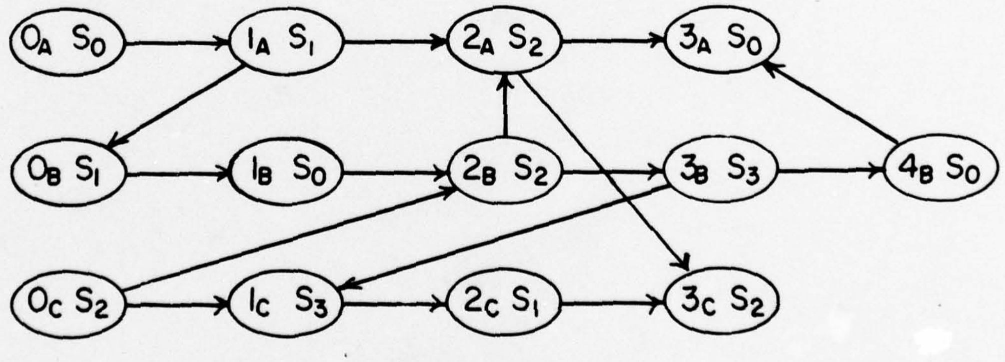
To see why LPF is of type PIP consider the pipeline of Figure 3.5.1a. Group together the computation steps of each function type according to their numbers, that is, all first steps in one group, all second steps in another group and so on. This grouping is indicated by large ovals in Figure 3.5.2a. If these groups are laid out according to the step numbers in an increasing order (as is done in Figure 3.5.2a), the edges representing the precedence will always be directed from left to right. In LPF, lower numbered steps have priority

	0	1	2	3	4
S_0	A	B		A	B
S_1	B	A	C		
S_2	C		AB	C	
S_3		C		B	

(a) Reservation table.

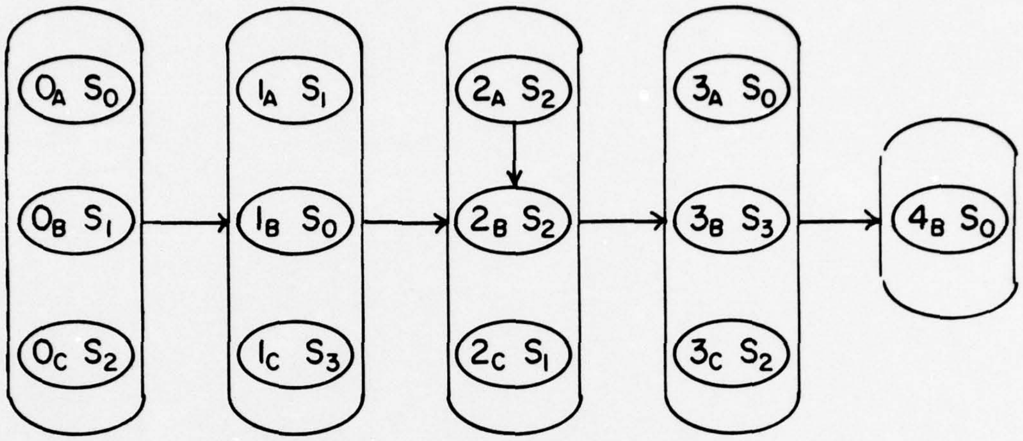


(b) Precedence graph.

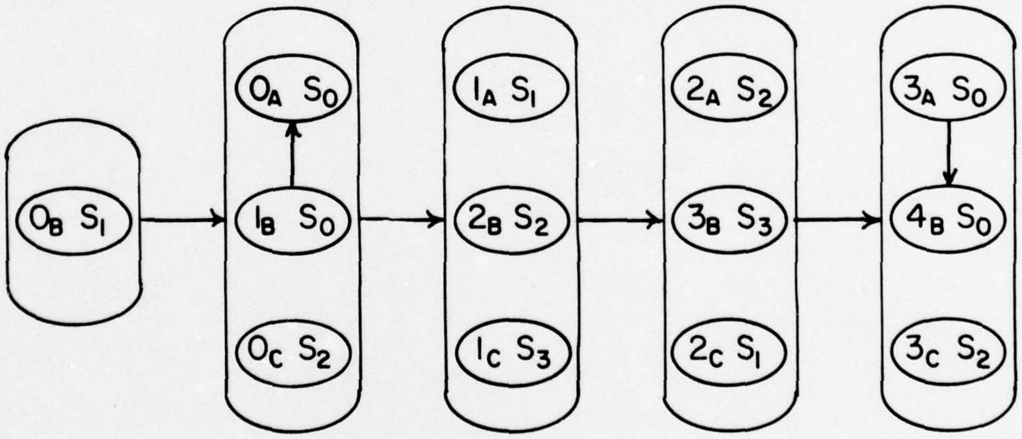


(c) Precedence-priority graph.

Figure 3.5.1. Precedence-priority graph of a multifunction pipeline.



(a) LPF priority.



(b) MWRF priority.

Figure 3.5.2. Precedence-priority graphs with LPF and MWRF priorities.

over higher numbered steps and therefore the edges representing the priority between distinctly numbered steps will also be directed from left to right. Thus the precedence and priority edges between the groups of steps cannot form a closed path. For steps within a group the only edges, if any, are priority edges (e.g. steps numbered 2 in Figure 3.5.2a). These edges do not form a cycle, because if they do then the priority among these steps is ambiguous and does not represent a valid priority scheme. Thus we have shown that the precedence-priority graph of a pipeline with LPF priority is acyclic. Therefore the LPF priority is of type PIP.

For MWRF priority we group the steps which are equally far from the last steps of their respective function types. Such a grouping is shown in Figure 3.5.2b for the same pipeline as above (Figure 3.5.1a). Using exactly the same arguments as those for LPF, we conclude that the precedence-priority graph is acyclic. Thus MWRF priority is also of type PIP.

To simplify the following discussion of PIP, we shall assign an integer called <u>precedence-priority-number</u> (ppn) to each node in the precedence-priority graph. The assignment of ppn's is done as follows. It is assumed that the priority in the graph is of type PIP.

<u>Algorithm 3.5.1</u>:

```
P1.  i ← 1;
P2.  collect all the nodes in the graph which do not have any pre-
     decessors; assign each of these nodes a ppn value equal to i;
```

P3. Remove the above nodes (i.e. nodes with ppn = i) and their outgoing arcs from the graph;

P4. $i \leftarrow i+1$; if no more nodes left in the graph then terminate else go to P2. □

For the graphs of Figures 3.5.1c, 3.5.2a and 3.5.2b the assignment of ppn's is shown in Figures 3.5.3a, b, and c, respectively.

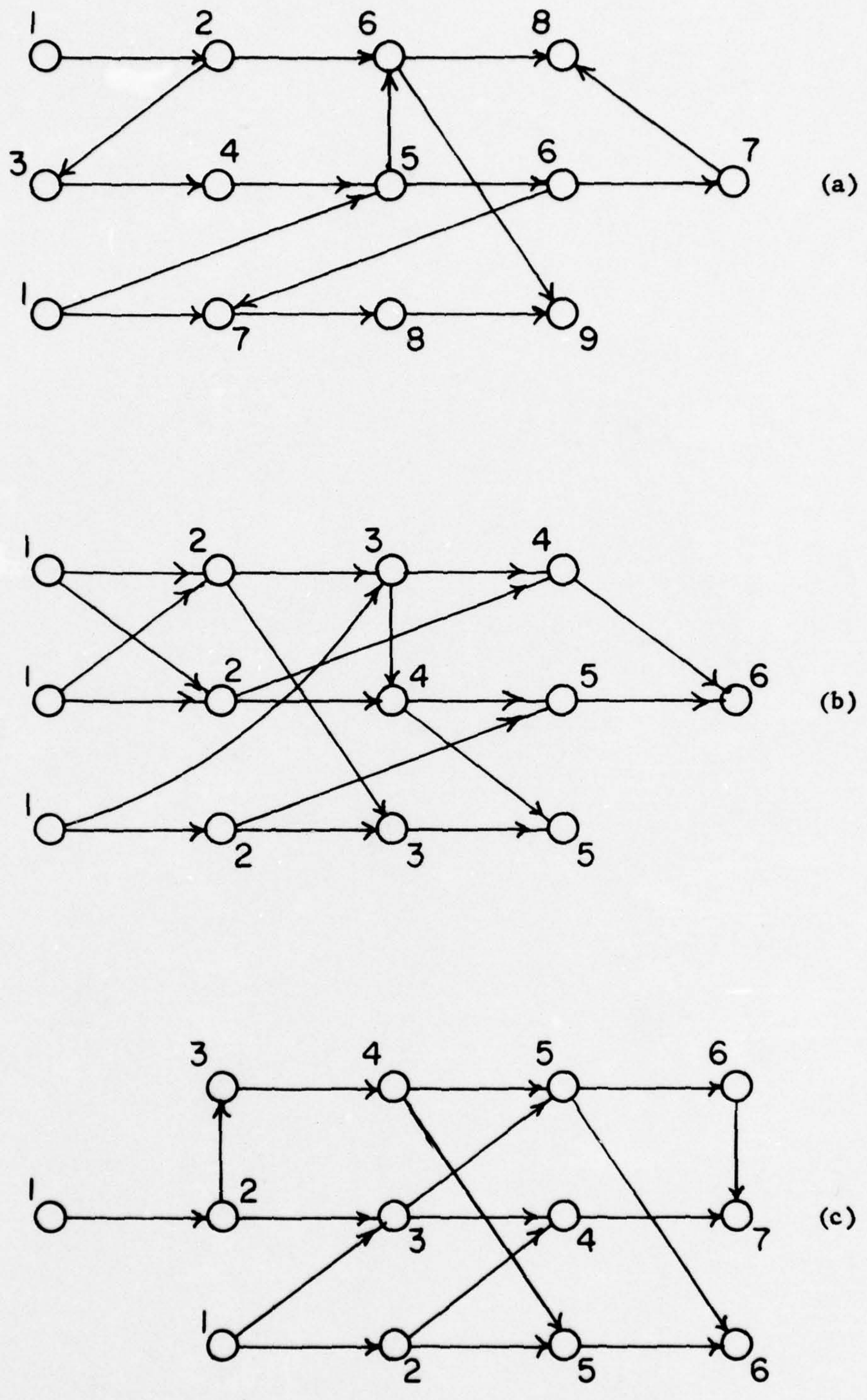
The ppn's assigned in the above manner satisfy the following properties.

Steps having the same ppn are unrelated by precedence as well as priority. Nodes with lower ppn values have higher priority. The flow of tasks in the pipeline is always from steps with lower ppn towards steps with higher ppn. These properties allow us to prove the following important theorem.

Theorem 3.5.1: In a buffered pipeline with PIP, the usage pattern of segments eventually (in finite time) becomes periodic with the period of the initiation cycle.

Proof: Let the computation steps of the pipeline be assigned the ppn values according to Algorithm 3.5.1. We shall prove the theorem by induction on ppn.

Basis step: Take the steps with ppn = 1. These steps have the highest precedence. The tasks arrive at these steps with the period p, the period of the initiation cycle. Furthermore, these steps have the highest priority and therefore the execution of these steps occur with the same period as the initiation cycle, namely, p.



<u>Figure 3.5.3</u>. **Examples of ppn assignment in precedence-priority graphs.**

Induction step: Assume that the execution of steps with $ppn < i$ is periodic with period p. We show that the execution of steps with $ppn = i$ eventually becomes periodic with period p.

The arrivals at the steps with $ppn = i$ are periodic with period p, because by hypothesis, all the predecessor steps (i.e., steps with $ppn < i$) are being executed with a period p. The available execution opportunities for the steps with $ppn = i$ are also periodic with period p for the same reasons. Furthermore, the activities of the steps with $ppn > i$ do not affect the execution of the steps with $ppn = i$. Therefore the execution of steps with $ppn = i$ eventually becomes periodic with period p.

Thus the execution of all steps eventually becomes periodic with period p. Q.E.D.

We shall say that a pipeline is in the steady state when all the computation steps are being executed periodically. The following corollary is merely an interpretation of the usage pattern being periodic with a period p. Recall here that an implementation of one buffer per computation step is assumed throughout this section.

Corollary 3.5.1.1: In a buffered pipeline with PIP the following holds during the steady state.

1. A task T_i arrives at some buffer at time $t \Rightarrow T_{i+n}$ arrives at the same buffer at time $t+p$. $\forall\ i \geq 0$.
2. A task T_i is processed by some segment during interval $\langle t,t+1\rangle$ for its kth computation step $\Rightarrow T_{i+n}$ is processed by the same segment during $\langle t+p,t+p+1\rangle$ for its kth step. $\forall\ i \geq 0$. □

The following lemmas and theorems lead to the establishment of bounds on queue size and wait times during the steady state of the pipeline. In all of the following, "a buffer of type X" means "a buffer at some computation step of a task of type X."

Lemma 3.5.2.1: During the steady state of a buffered pipeline with PIP, no more than n_X tasks of type X arrive at and no more than n_X tasks of type X depart from a buffer of type X, during any time interval $\langle t,t+p \rangle$.

Proof: Consider the buffer of the first step of function X. Exactly n_X tasks arrive at this buffer during any interval $\langle t,t+p \rangle$. In the steady state, from Corollary 3.5.1.1, if more than n_X tasks leave the buffer during some interval $\langle t',t'+p \rangle$ then more than n_X tasks must leave the same buffer during $\langle t'+p,t'+2p \rangle$,$\langle t'+2p,t'+3p \rangle$,.... and so on. However, this departure rate cannot go on indefinitely because more tasks would leave the buffer than would arrive. Since the buffer can only contain a finite number of tasks to start with, the steady state assumption would be violated. Thus no more than n_X tasks arrive at or leave the first buffer during any interval $\langle t,t+p \rangle$ in steady state.

Whatever leaves the first buffer goes to the second buffer. Thus the second buffer receives no more than n_X tasks during any interval $\langle t,t+p \rangle$. Similarly, no more than n_X tasks leave the second buffer during any interval $\langle t,t+p \rangle$. By finite induction, the theorem is proved. Q.E.D.

When the above lemma is applied simultaneously to all the buffers of a segment the following lemma results.

Lemma 3.5.2.2: During the steady state of a buffered pipeline with PIP, the segment i receives no more than $\sum_x m_{iX}n_X$ tasks and it processes no more then the same number of tasks during any interval $\langle t, t+p \rangle$. □

Lemma 3.5.2.3: During the steady state of a pipeline with PIP and a workload < 100%, every buffer at segment i is empty for at least one unit time interval during any interval $\langle t, t + \sum_X m_{iX}n_X \rangle$.

Proof: If a buffer associated with segment i remains nonempty for $\sum_X m_{iX}n_X$ consecutive time units then segment i is busy for at least $\sum_X m_{iX}n_X + 1$ consecutive time units. The workload being < 100% implies that $\sum_X m_{iX}n_X < p$; i.e., $\sum_X m_{iX}n_X + 1 \leq p$. Thus segment i remains busy for $\sum_X m_{iX}n_X + 1$ units during some interval $\langle t, t+p \rangle$. This statement contradicts Lemma 3.5.2.2. Hence, the buffer must remain empty for at least one time unit during any interval $\langle t, t + \sum_X m_{iX}n_X \rangle$. Q.E.D.

The above lemma is our assurance that irrespective of the priority within a buffer, all tasks will be processed when workload is < 100%. Thus for example we may use a LIFO-buffer without the fear that a task may remain in some buffer indefinitely. However, notice that we have assumed a workload strictly < 100%, a deviation from our standard assumption of workload $\leq$ 100%. Unfortunately these results are not fully applicable with 100% workload as discussed later in this section.

Theorem 3.5.2: During the steady state of a buffered pipeline with PIP and a workload < 100%, the number of tasks in any buffer of type X is always $\leq n_X$.

Proof: Consider a particular buffer of type X. By Lemma 3.5.2.3 the buffer is empty for at least one time unit during any interval $\langle t, t + \sum_X m_{iX} n_X \rangle$. Suppose this unit interval during which the buffer is empty is $\langle t', t'+1 \rangle$. Now due to the steady state, the buffer is also empty during $\langle t'+p, t'+p+1 \rangle$. Moreover by Lemma 3.5.2.2 no more than n_X tasks can arrive during the interval $\langle t', t'+p \rangle$. Thus no more than n_X tasks can accumulate in the buffer. Q.E.D.

Theorem 3.5.3: During the steady state of a buffered pipeline with PIP and a workload < 100%, the following bounds on wait times hold:

1. The maximum wait time of a task for a computation step at segment i is

$$\leq \left(\sum_X m_{iX} n_X\right) - 1.$$

2. The expected wait time of a task for a computation step at segment i is

$$\leq \frac{1}{2} \sum_X m_{iX} n_X - \frac{\sum_X m_{iX} n_X^2}{2 \sum_X m_{iX} n_X} .$$

Proof: 1. Directly follows from Lemma 3.5.2.3.

2. Once we establish that all the buffers at segment i remain empty simultaneously for one time unit during some interval $\langle t, t + \sum_X m_{iX} n_X \rangle$, the rest of the proof is identical to the proof (2) of Theorem 3.3.3.

Consider the lowest priority buffer at segment i. When this buffer becomes empty due to selection of the last task in the buffer for execution, it is implied that all higher priority buffers must be empty. Thus all buffers become empty simultaneously at least once during any period $\langle t,t+p\rangle$. From this instant on all the arguments in the proof of Theorem 3.3.3 apply and therefore we omit here the rest of the proof. Q.E.D.

As stated earlier in Theorem 3.4.1, the bounds on the queue length and the expected wait time are independent of the priority used within a buffer. However, the bound on the maximum wait time does depend on this priority. In our proof of (1) of Theorem 3.5.3, we did not use any specific priority within a buffer. A tighter bound may be obtained for some specific priority.

Take for example FIFO-buffer. Let the lowest priority buffer for the segment i be of type Y. Clearly, the worst case delay can occur only for the tasks in the lowest priority buffer. Suppose a task arrives at this buffer when it is empty. Since the buffer is first-in-first-out, this task can only be delayed by the tasks from other buffers. During every period $\langle t,t+p\rangle$ a total of $\sum_X m_{iX}n_X$ steps are executed on segment i, of these n_Y tasks are taken from the lowest priority buffer. Therefore the task under consideration can be delayed for at most $\sum_X m_{iX}n_X - n_Y$ time units. It can be shown that even if a task arrived at the buffer when the buffer was not empty, the task cannot be delayed more than $\sum_X m_{iX}n_X - n_Y$ time units.

For LIFO-buffer the bound can be shown to be the same as in (1) of Theorem 3.5.3. For other priorities within the buffer the bound would lie between the bounds for FIFO-buffer and LIFO-buffer.

All the bounds shown so far are proven under two restrictions. One is that the workload be strictly less than 100%. If the workload is assumed to be equal to 100% then the arguments in the proof of Lemma 3.5.2.3 are no longer applicable to the buffers with the least priority on each segment i for which $\sum_X m_{iX} n_X = p$. However, the result of that lemma is still valid for other buffers and hence the bounds of Theorems 3.5.2 and 3.5.3 are also valid for these buffers.

Another restriction that we imposed in the above theorems is that of steady state. We have put considerable effort in investigating the transients. As yet, we do not have any concrete results to report. However, we have gained considerable insight into the problem and this insight has lead us to believe the following two conjectures.

<u>Conjecture 3.5.1</u>: A buffered pipeline with PIP and workload $\leq$ 100% reaches a steady state within the number of initiation cycle periods equal to the value of the largest ppn in the precedence-priority graph of the pipeline. □

<u>Conjecture 3.5.2</u>: In a buffered pipeline with PIP, the bounds of Theorems 3.5.2 and 3.5.3 are valid during the transient and steady state and for all workloads $\leq$ 100%. □

The above conjecture can be shown to be true in a very special case namely, single function pipelines with constant latency cycles. In a single function pipeline PIP is the same as LPF and MWRF. Therefore by Theorem 3.4.3 the task flows are identical with LIFO-global and PIP in this special case. Thus the results concerning LIFO-global priority directly apply to PIP as stated below.

Theorem 3.5.4: In a buffered single function pipeline with PIP and a constant latency cycle with workload $\leq 100\%$ the following bounds are satisfied.

1. Maximum number of tasks in a buffer is ≤ 1.
2. Maximum wait time of a task for a computation step at segment i is $\leq (m_i - 1)$.
3. The expected wait time of a task for a computation step at segment i is $\leq \frac{1}{2}(m_i - 1)$.

Proof: 1. Follows from the proof of Theorem 3.4.3 namely, that no two tasks can be together in a buffer at any instant.

2. By Theorem 3.4.3 the bound in (1) of Corollary 3.3.3.1 applies, namely, $(m_i n - 1)$. For constant latency cycles n is 1 and therefore the bound is $(m_i - 1)$.

3. For the same reasons as above the bound (2) of Corollary 3.3.3.1 applies which at $n = 1$ is $\frac{1}{2}(m_i - 1)$. Q.E.D.

Regarding the tightness of the bounds of Theorems 3.5.2 and 3.5.3, the same comments as on the bounds of LIFO-global priority apply. Specifically, the bounds are the best possible with the information used, but in general, the actual maximum queue size and wait time may be considerably less than the stated bounds.

3.6. Concluding Remarks

The principal result of this chapter is that one can always obtain the maximum throughput of a pipeline with the use of internal buffers. Moreover, one has complete freedom in choosing an initiation cycle. We have shown that any initiation cycle with a workload $\leq$ 100% can be processed by a pipeline with internal buffers using either LIFO-global priority or any other priority of type PIP. Two priorities of interest, namely, LPF and MWRF were shown to be of type PIP.

We also presented the bounds on queue size and wait time for LIFO-global and PIP. Although for PIP we were able to prove these bounds only when we had assumed that workload was < 100% and the *pipeline was in steady state.* The bounds were found to be the best possible with the information used. Some examples tried by us have shown that the actual queue size and wait time are likely to be considerably less than the stated upper bounds. For a particular pipeline we suggest simulation to determine the actual bounds. At least for LIFO-global and PIP we know by experience that the simulation is required only for a short time because the steady state is reached in a short time. In fact, in most cases hand computation is sufficient to trace out the task flows until the flow becomes periodic.

None of the above comments are applicable to FIFO-global, MPF and LWRF priorities. Nevertheless, we have not yet found any example in which the steady state was not reached. Although not yet proven, we believe that the queues are bounded and that the maximum throughput is possible with any cycle.

For guidance in choosing a cycle with any of the above priorities, the same comments can be made as in Section 2.11. Namely, that a cycle with small period and evenly spaced task initiations is likely to produce small queue sizes and small wait times.

In this chapter we did not discuss how fast one can switch from one cycle to another. As far as PIP, FIFO-buffer is concerned the problem is not serious except that we cannot predict the queue bounds and wait times precisely. While in LIFO-global, the possibility exists that some tasks from the previous cycle may never get processed. Therefore we suggest that any such switching must be done with care, and a simulation should be carried out to predict any such unusual behavior. Of course, one can always use the conservative approach and let the pipeline become empty before another cycle is started. This practice can result in a degradation of throughput if the switching of cycles is frequent.

Results of this chapter are applicable only in a well-defined environment, such as in a vector machine where vector instructions can be conveniently converted into initiation cycles. When the switching of cycles is frequent or when the arrivals are random, or unpredictable, the buffers are still useful in resolving the collisions. The next chapter deals with buffered pipelines with random arrivals.

Chapter 4

INTERNAL BUFFERS WITH RANDOM ARRIVALS

4.1. Introduction

In this chapter, we investigate pipelines with internal buffers and random rather than periodic initiations. In Chapter 1 we indicated the need for processing random sequences of tasks in the order their arrival. The arithmetic units of the existing pipeline computers (e.g. TI-ASC and CDC-STAR) use a very conservative approach to this problem. In these machines only those instructions are overlapped which are of the same type, e.g., a sequence of multiplies. A sequence of distinct instructions is executed without any overlap, that is, one instruction at a time. Since a program of general nature normally does not contain many sequences of consecutive instructions of the same type, these machines produce a throughput considerably below the possible maximum throughput. Even "intelligent" compilers for such machines which try to compile programs into "vector instructions" (i.e., sequences of operations of the same type) find substantial portions of code non-vectorizable (i.e., "scalar"). The only other proposed control strategy for scalar instruction overlap is the greedy strategy of Davidson [DAV71] and Thomas [THO74]; which even though superior to the existing strategies, still produces a throughput well below the possible maximum throughput, as we shall see later in this chapter. Here we propose yet another alternative, namely, the use of internal buffers.

We shall experimentally evaluate the effectiveness of internal buffers and greedy control in this chapter. We shall investigate by simulation only those priority schemes which are based on the processing state because of their simpler implementation than FIFO-global and LIFO-global as was described in Section 3.4. Moreover, FIFO-global is very similar to MPF and LWRF as indicated by Theorem 3.4.1. LIFO-global cannot be used, because there exists a possibility that some tasks may never get processed or they may be forced to wait for an unusually long time. For the same reasons LIFO-buffers are ruled out. The buffering scheme is assumed to be one FIFO buffer per computation step.

The effect of the following parameters on performance will be investigated:

1. Buffer size
2. Workload
3. Priority scheme.

All buffers except the input buffers are assumed to have the same size. The input buffer size is assumed to be unbounded. Some modification of the priority schemes is necessary because of the finiteness of the buffers inside the pipeline. Before we describe the modification recall that an equivalent of three step action takes place at every time instant (Section 3.2). First, all tasks leave the segments and arrive in their next buffers; second, the priority control selects the tasks with highest priorities and third, the selected tasks are forwarded to the appropriate segments. The modification imposed is that the priority control considers only those buffers whose successor buffers are not full at the current time instant. Thus a task will not leave a buffer if its successor buffer is full even if the segment may remain idle. This modification guarantees that a segment can always deliver a task to the next buffer without the possibility of an overflow.

Note that the priority control considers only those buffers whose successor buffers are not full at the current time instant. It is possible to avoid an overflow by considering only those buffers whose successor buffers are not going to be full at the next time instant. However, the latter implementation is far more complicated than the previous one since the decision process has to occur at a global level rather than at local level. Moreover, this decision process is sequential rather than parallel since the selection of a task from a buffer might cause a selection of another task from its predecessor buffer and a task from pre-predecessor buffer and so on.

We shall adhere to the modification suggested originally. The following priority schemes will be studied. A * indicates that the priority as explained in Section 3.4 is modified with respect to successor buffers in the manner described above:

1.	MPF*:	Modified Most Processed First
2.	LWRF*:	Modified Least Work Remaining First
3.	LPF*:	Modified Least Processed First
4.	MWRF*:	Modififed Most Work Remaining First
5.	RANDOM*:	Modified Random.

In RANDOM, the priority is randomly assigned among buffers. Like all other priorities we have considered, in RANDOM also we assume that once the priority is assigned, it remains fixed. RANDOM serves as a basis of comparison for the performance of other priority schemes.

The following assumptions are made regarding task arrivals:

1. At most one task arrives at each buffer in a unit time interval.
2. Arrivals at the pipeline are random according to a binomial distribution and independent of the processing rate.
3. The mean arrival rate of tasks of each function type is fixed.

The performance will be judged based on the following parameters:

1. Segment utilization.
2. Expected average wait time of a task from arrival to termination, w_1:

   w_1 = (time of termination) - (time of arrival) - (time to execute a task without buffers).
3. Expected average wait time of a task from initiation in the pipeline to termination, w_2.

   w_2 = (time of termination) - (time of initiation) - (time to execute a task without buffers).

The throughput of a pipeline is proportional to segment utilization when the buffers are assumed finite in size. A 100% utilization of some segment corresponds to the maximum possible throughput.

The second parameter w_1 above gives an indication of the turn around time of a task. A small turn around time is a necessity in many real-time applications.

The third parameter w_2 differs from the second parameter in that the wait incurred in the input buffer is not included. A small w_2 is preferred in some situations for the following reasons.

A large w_2 implies a large number of tasks inside the pipeline. These tasks are partially processed and therefore if the normal processing is to be suspended for any reasons (e.g., due to an interrupt) all the partially processed tasks either must be completed or restored to their original state. Thus a large number of partially processed tasks implies a slow response to say an interrupt. Thus a small w_2 is desirable in some cases.

4.2. Experimental Investigation

Analytical results for internally buffered pipelines with random arrivals are very difficult to obtain. Markov analysis can be applied. However, even for a small pipeline this analysis can become quite complex. Markov analysis was applied to a few trivial pipelines for the sole purpose of verifying the validity of our simulation techniques by comparing the results of these two methods.

For simulation the pipelines were created with random task flows. For a given number of segments and a given number of function types, each type of task flow was determined by a random sequence of integers, each integer taken between 1 and the number of segments with equal probability. The length of the sequence (which is the same as the execution time of the task) was also taken arbitrarily, however taking care that it is not too long and thus not too complex computationally.

The pipelines of Table 4.2.1, created in the above manner were used for simulation. The pipeline designation j.k stands for a pipeline with j function types and k segments. The flow of each function type is given by a sequence of integers, where each integer stands for a segment number.

In each case the simultion was done for a fixed duration and no attempt was made to continue the simulation until a steady state was reached because in theory, for some cases a steady state may not exist, and even if it did exist it could take an unreasonably long time to reach it. However, for most practical purposes a steady state was reached in many cases during the simulation period, especially at low and medium workloads. The following is a discussion of the simulation results.

4.3. Observations on the Experimental Results

We have presented in Table 4.3.1 some compacted results of our simulation. Each pipeline of Table 4.2.1 was simulated for three different workloads. In Table 4.3.1, the column marked W indicates the workload, where HI, MED and LOW are respectively 100%, 75% and 50% workload. The column marked BUF indicates the buffer size in each run. U% is the maximum utilization of any segment in the pipeline and w_1 and

TABLE 4.2.1

Pipelines used for simulation

Pipeline	Flow of Tasks
1.4	F1: 4 1 4 1 3 1 4 2 2 2 1
4.4	F1: 4 3 4 4
	F2: 1 4 1 3 3 1 4 2 2
	F3: 2 2 1
	F4: 3 1 2 2 4 4 2 1
4.8	F1: 5 1 2 5 7 7 4 5 2 1 1 6 3 8 6 7 5 6
	F2: 8 2 6 5 1 6 6 5 6 6 1 4
	F3: 6 5 8 3 1 3 8 8 5 7
	F4: 3 6 8 3 7 2 5
4.16	F1: 8 2 6 8 2 7 13 7 3 4 6 14 2 11 6 13 9 2 5 13 5
	F2: 1 16 13 2 11 10 11 9 2 1
	F3: 2 8 6 3 13 16 12 4 4 7 3
	F4: 2 6 15 8 2 15 12 11 2 1 8 9
8.4	F1: 2 1 2 2 1 2 4 2 1 1 2
	F2: 1 2
	F3: 4 2 1 4 3 3 1 1 2 3 1 1 3 2 2 2 4 1 2 1 3 4 2 4 1 4 3 2 1
	F4: 1 1 3 1 1 2 4 3 4 3 4
	F5: 1 2 2 1 1 3 4 4
	F6: 2 1 4 1 2 1
	F7: 2 1 4 3 1 4 4 3 4 4 3 3 2 1
	F8: 3 2

w_2 are the wait times as defined earlier in Section 4.1. Figures 4.3.1 through 4.3.4 graphically illustrate the effect of various parameters on performance for a "typical" example. The pipeline 4.4 of Table 4.2.1 was used for this example. The data of these figures do not appear in Table 4.3.1. The example is typical in the sense that a similar effect can also be observed for most examples in Table 4.3.1. However, it should be pointed out that anomalies always exist which do not exhibit the typical behavior.

The observations below can be made from the data of Table 4.3.1 and Figures 4.3.1 through 4.3.4. As described before, w_1 refers to the expected average wait time of a task from arrival to termination and w_2 refers to the expected average wait time of a task from its initiation in the pipeline to its termination.

1. Segment Utilization (See Table 4.3.1): The maximum segment utilization closely follows the workload in most cases except in a few cases when the workload is high ($\sim$ 99%) and buffer size is small. A possible explanation for this is derived from the modification of the priority strategies. Recall that a task may not leave a buffer if its successor buffer is full even if the segment is available. A large number of buffers are likely to be full if their size is small and workload is high; thus forcing some segments to become idle some of the time. In this type of situation it is apparent that an increase in buffer size results in higher utilization of the pipeline. The utilization is otherwise somewhat insensitive to the buffer size used. It is also insensitive to the priority used.

Table 4.3.1. Simulation results.

PIPE	W	BUF	RANDOM*			LPF*			MWRF*			MPF*			LWRF*			GREEDY
			U%	w1	w2	U%	w1	w2	U%	w1	w2	U%	w1	w2	U%	w1	w2	U%
	H	2	96.5	35	21	94.6	74	49	94.6	74	49	96.6	28	5	96.6	28	5	
	I	3	96.6	39	31	96.3	73	65	96.3	73	65	96.6	28	8	96.6	28	8	85.4
		∞	96.6	58	58	96.6	105	105	96.6	105	105	96.6	28	27	96.6	28	27	
	M	2	73.7	7	7	73.6	14	14	73.6	14	14	73.7	4	2	73.7	4	2	
1.4	E	3	73.7	8	8	73.7	14	14	73.7	14	14	73.7	4	3	73.7	4	3	69.6
	D	∞	73.7	9	8	73.7	15	15	73.7	15	15	73.7	4	4	73.7	4	4	
	L	1	49.4	2	2	49.4	5	5	49.4	5	5	49.4	2	1	49.4	2	1	
	O	2	49.5	3	3	49.5	5	5	49.5	5	5	49.5	2	1	49.5	2	1	49.9
	W	∞	49.5	3	3	49.5	5	5	49.5	5	5	49.5	2	1	49.5	2	1	
	H	2	88.9	110	19	88.2	128	31	97.5	49	29	97.8	84	4	90.3	112	10	
	I	3	90.3	106	28	92.0	103	48	98.1	48	37	98.1	50	5	91.9	94	13	56.5
		∞	97.5	59	56	96.3	95	95	98.2	83	83	98.0	40	6	95.7	59	50	
	M	2	74.1	6	5	74.1	8	8	74.1	7	7	74.1	5	3	74.0	5	3	
4.4	E	3	74.1	6	5	74.1	8	8	74.1	8	8	74.1	5	3	74.1	5	3	51.4
	D	∞	74.1	6	5	74.1	8	8	74.1	8	8	74.1	5	3	74.1	5	3	
	L	1	48.4	2	1	48.4	2	2	48.4	2	2	48.3	2	1	48.4	2	1	
	O	2	48.4	2	1	48.4	2	2	48.4	2	2	48.3	2	1	48.4	1	1	44.0
	W	∞	48.4	2	1	48.4	2	2	48.4	2	2	48.3	2	1	48.4	1	1	
	H	2	94.8	54	33	88.4	126	60	92.6	100	68	94.2	129	9	96.4	37	11	
	I	3	95.6	54	44	88.8	126	89	94.6	94	87	94.0	120	12	97.0	35	13	49.5
		∞	97.1	61	60	97.2	103	103	97.0	171	171	94.8	111	62	97.0	34	19	
	M	2	72.7	5	3	72.7	6	6	72.7	6	6	72.6	6	3	72.7	4	3	
4.8	E	3	72.7	5	4	72.7	6	6	72.7	7	7	72.6	6	4	72.7	5	4	54.2
	D	∞	72.7	5	3	72.7	7	7	72.7	8	8	72.6	6	4	72.7	5	4	
	L	1	50.6	3	2	50.5	3	3	50.5	4	3	50.5	3	2	50.5	3	2	
	O	2	50.5	3	3	50.5	3	3	50.5	4	4	50.5	3	2	50.5	3	2	41.8
	W	∞	50.5	3	3	50.5	3	3	50.5	4	4	50.5	3	2	50.5	3	2	
	H	2	99.2	61	19	98.8	78	54	99.3	83	48	96.0	216	5	98.6	58	8	
	I	3	99.3	64	21	99.4	86	77	99.4	86	60	99.0	190	6	99.1	53	10	55.2
		∞	99.4	63	19	99.2	136	136	99.3	78	78	99.4	184	6	99.2	58	8	
	M	2	72.8	4	4	72.8	5	5	72.8	5	5	72.8	4	2	72.8	4	3	
4.16	E	3	72.8	5	4	72.8	5	5	72.8	5	5	72.8	4	2	72.8	4	3	54.4
	D	∞	72.8	5	4	72.8	5	5	72.8	5	5	72.8	4	3	72.8	4	3	
	L	1	50.1	2	1	50.1	2	1	50.1	2	1	50.1	2	1	50.1	2	1	
	O	2	50.1	2	2	50.1	2	2	50.1	2	2	50.1	2	1	50.1	2	1	49.6
	W	∞	50.1	2	2	50.1	2	2	50.1	2	2	50.1	2	1	50.1	2	1	
	H	2	96.6	122	9	98.2	28	28	98.8	61	46	98.3	47	3	97.8	19	12	
	I	3	97.3	110	10	98.6	20	20	99.1	64	47	98.7	43	4	98.2	19	14	61.4
		∞	98.7	95	12	98.7	15	15	99.4	93	66	98.6	45	6	98.2	19	16	
	M	2	72.7	13	10	73.0	18	18	72.9	22	22	72.5	12	8	71.4	9	8	
8.4	E	3	72.8	12	10	73.0	18	18	72.9	23	23	72.6	13	9	72.7	9	8	48.8
	D	∞	72.8	12	10	72.7	19	19	72.9	28	28	72.7	13	9	72.7	9	8	
	L	1	50.8	6	5	50.8	7	7	50.8	9	8	50.8	6	4	50.8	5	4	
	O	2	50.8	6	5	50.8	8	8	50.8	10	10	50.8	6	4	50.8	5	4	46.6
	W	∞	50.8	6	5	50.8	8	8	50.8	11	10	50.8	6	4	50.8	5	4	

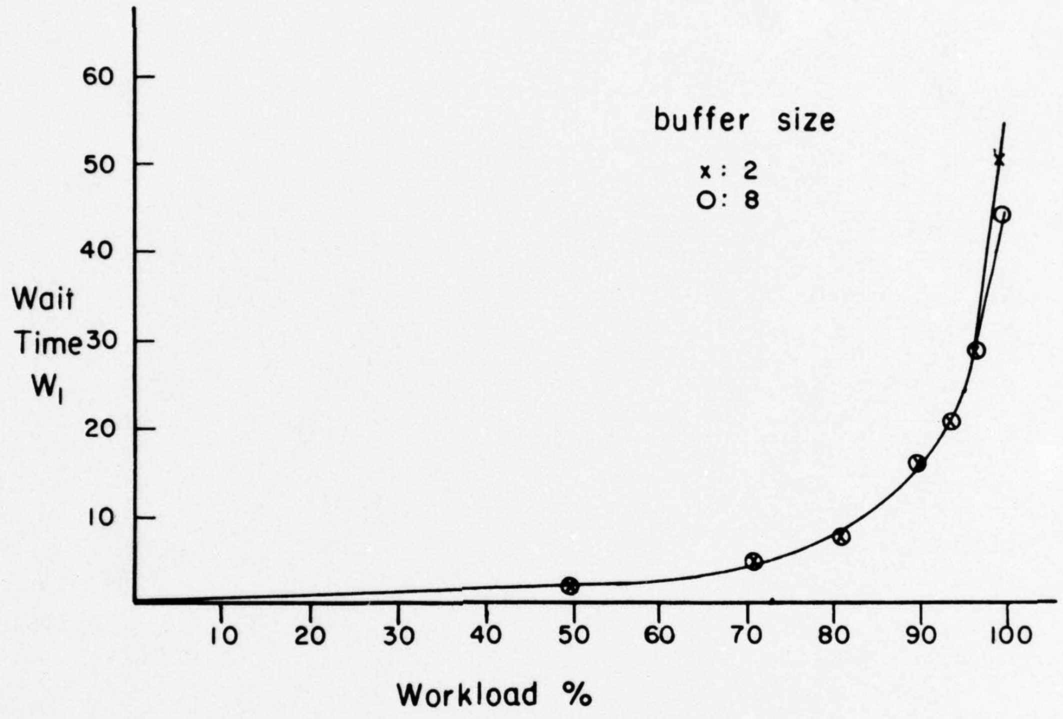


Figure 4.3.1. Plot of expected wait time w_1 vs. workload.

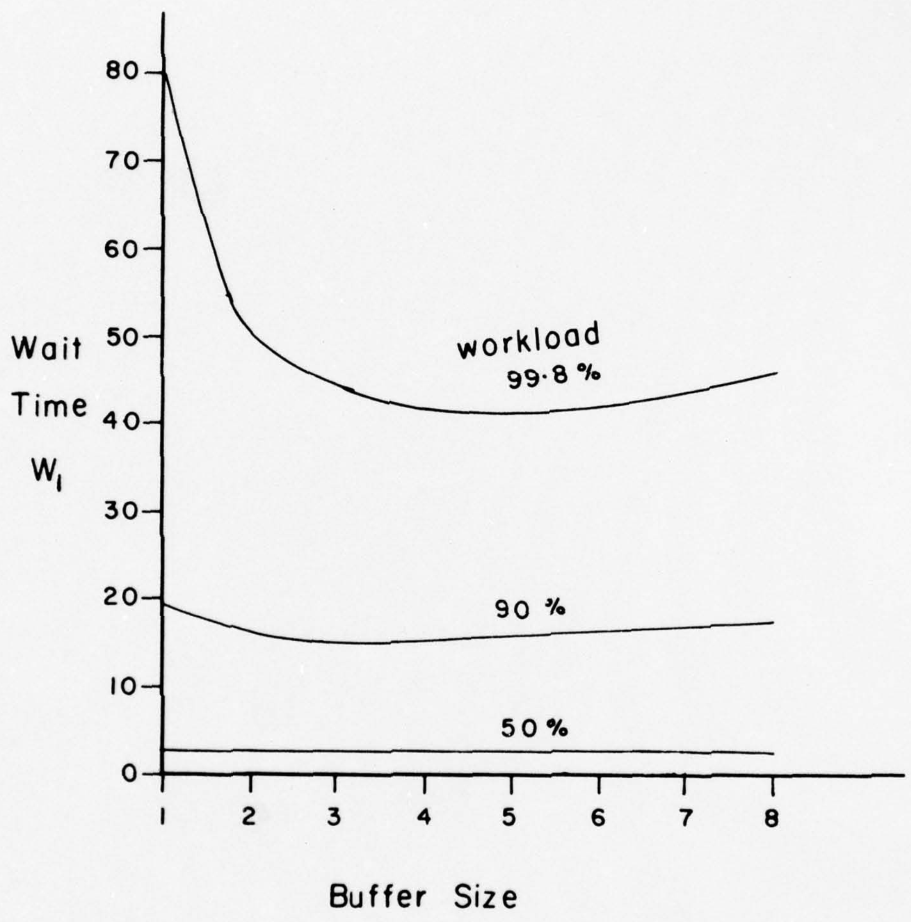


Figure 4.3.2. Plot of expected wait time w_1 vs. buffer size.

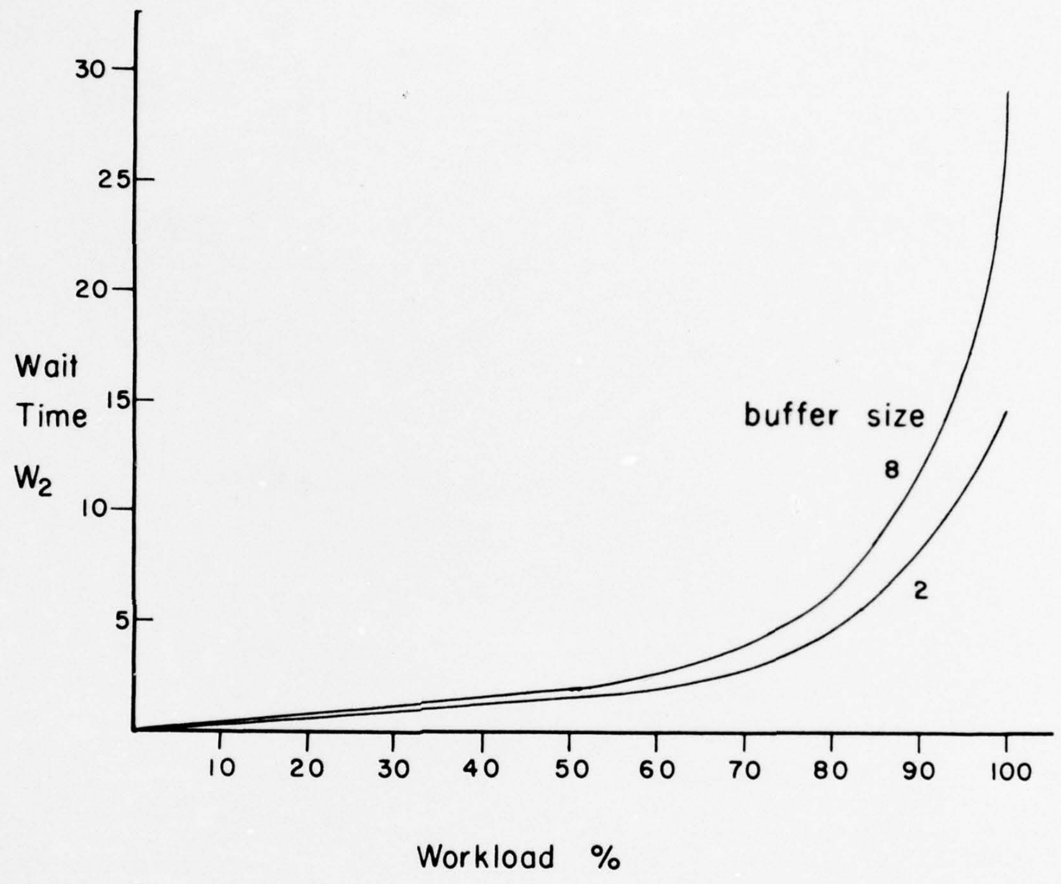


Figure 4.3.3. Plot of expected wait time w_2 vs. workload.

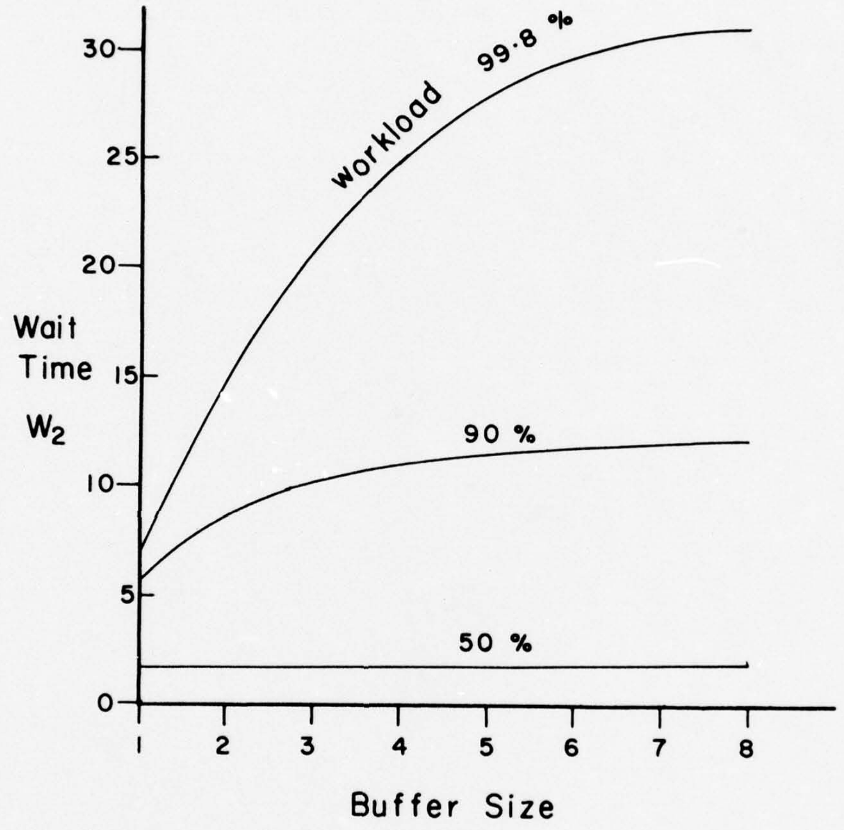


Figure 4.3.4. Plot of expected wait time w_2 vs. buffer size.

For comparison we have also simulated the greedy strategy. The maximum segment utilization achieved in each case is listed in Table 4.3.1. Excepting the single function pipeline, the maximum utilization achieved in all examples is in the neighborhood of 55 to 60%. Since the throughput is proportional to segment utilization we see that with greedy strategy, the throughput remains considerably below the theoretical maximum. In contrast to this, internally buffered pipelines achieved a throughput very near to the theoretical maximum in most of the examples that we tried.

2. The effect of workload on wait time (see Figures 4.3.1 and 4.3.3): It is quite apparent that irrespective of the buffer size and priority, the wait times w_1 and w_2 are relatively insensitive to workload in the low and medium load region and both these wait times rise rapidly in the 90% to 100% workload region.

   The reason for this is not hard to see. At high workloads the possibility of task collisions increases because at any time instant a pipeline is likely to have more tasks present at high workloads than at low workloads. Furthermore, when the number of tasks in the pipeline is high, one collision can trigger multiple collisions, thus further adding to the wait time.

3. The effect of buffer size on wait time w_1 (see Figure 4.3.2): The expected average wait time of a task is affected very little by change in buffer size beyond the size of 2 in the medium workload (50% to 90%) region and is almost unaffected by buffer sizes in the low workload ($<$ 50%) region. In the very high workload region, with

very small buffers the increase in buffer size would normally result in a decrease in wait time w_1. This effect is similar to the effect on segment utilization and for the same reasons. It is interesting to note a dip in the curves for 99.8 and 90% workloads in Figure 4.3.2. The dip is not very significant but it is still somewhat puzzling. We do not have a reasonable explanation for that dip.

4. The effect of buffer size on wait time w_2 (see Figure 4.3.4): From Table 4.3.1 and Figure 4.3.4 it is evident that w_2, the expected average wait time of a task inside a pipeline initially increases with the increase in buffer size and then approaches a constant value. This happens because when the buffers are very small, most of the tasks accumulate in the input buffer and thus a large portion of total wait time is incurred in the input buffers. Therefore w_2 remains small. Any increase in the buffer size results in reducing the number of tasks in the input buffers and increasing the wait time w_2. Once the buffer size is sufficiently high, buffers are rarely full and thus any increase in buffer size has very negligible effect on the task flow. Hence the wait time w_2 remains unaffected in this region.

5. The effect of priority (see Table 4.3.1): In most cases the smallest wait times occur with MPF* and LWRF* priorities; slightly higher wait times occur with RANDOM* priority and considerably higher wait times occur with LPF* and MWRF* priorities. The

following explanation can be given for this effect when the buffers are assumed finite.

In MPF* or LWRF* priorities, high priority buffers remain empty most of the time. The same is not necessarily true in LPF* or MWRF* priority. Consider what happens when a low priority buffer is full. By the modification imposed on priorities with finite buffers, a task cannot leave a buffer if its successor buffer is full. Thus a high priority buffer may become full because of its successor buffer being full. This effect may be propagated to the highest priority buffer. Thus even the highest priority buffer may not be empty some of the time. In the worst case it may happen that most of the buffers remain nearly full, thus resulting in a very high wait time. This argument, of course cannot be applied when the buffers are assumed infinite or when the workload is low. When the workload is low, the differences in wait times for various priorities are negligible. While at high workloads and infinite buffers the data of Table 4.3.1 does not conclusively indicate the superiority of one priority over others.

4.4. Concluding Remarks

We observed that a workload very near to 100% can be executed with internal buffers. In other words a throughput very near to the theoretical maximum can be attained. Moreover, the choice of priority is not very critical as far as throughput is concerned. However, if small wait times are preferred then the MPF* and LWRF* priorities were found to be superior.

It should be noted here that ours is an open loop analysis. In a closed loop model the arrival rate of tasks will depend on the processing rate and the wait times. Normally this dependency will have a stabilizing effect on the wait time. For example in a computer with a pipelined arithmetic unit, an increase in wait time in the arithmetic unit results in a reduced rate of arrivals of instructions, which in turn results in a lower wait time. The throughput, however, will also decrease. We did not attempt a closed loop analysis since it is highly problem dependent and since we are considering pipelines in general.

Although we shall not present any specific data it should be pointed out that the priority can be "fine-tuned" to advantage. We suggest the following fine-tuning procedure, which we have found very effective in our limited experience.

Start with either MPF* or LWRF* priority and do a simulation with appropriate arrival rates. Gather the individual wait time and queue size statistics of <u>each</u> buffer. Increase the priority of those buffers in which a disproportionately large amount of wait time is incurred and those buffers which remain full most of the time. Decrease the priority of those buffers which remain empty most of the time. After the priority is changed conduct the simulation again and readjust the priority in the above manner. Since this is somewhat of a trial-and-error procedure and since the results are not always predictable, several runs might be necessary. In a similar manner it is also possible to come up with an "optimum size" of buffer for each step.

We also simulated the greedy strategy in this chapter. We observed that in most cases a workload of up to 55 to 60% can be executed. Thus the maximum throughput achievable with greedy strategy is considerably below the throughput with internal buffers. Even though the greedy strategy compares poorly with internal buffers in throughput, it is not necessarily less cost-effective than internal buffers. We shall make some comments regarding cost-effectiveness in the next chapter.

Chapter 5

DESIGNING OF PIPELINES

5.1. Introduction

In this chapter we introduce some concepts for designing a pipeline. We shall describe various options a designer can exercise to meet his goals for performance and cost. The number of available options is so large that it is practically impossible to formalize the complete design procedure. However, some formalization does exist at a few steps in the design procedure. In a multistep procedure where each design step in itself is a major optimization problem, we can just hope that a design conceived by locally optimizing at each step is close to the global optimum. A designer must make judicious decisions at various points in design to cut down on the number of possible options and make the problem more manageable. We propose here a top-down design procedure. At every point in the design we try to achieve a local optimum. To bring the solution closer to the global optimum some backtracking must be done. How much effort should be spent on backtracking, depends on the criticality of the design requirements, for example, cost, execution delay, throughput, etc. In the following, we have relied heavily on illustrations to discuss each design step.

5.2. A Design Methodology

We start out with the given function or task which is to be pipelined. At this point we must have a specific goal, namely, the throughput which must be met or exceeded, the cost which must not be exceeded and the maximum allowable execution delay of each task.

As a running example let the function to be pipelined be $(a_0 + a_1x + a_2x^2)\cdot(b_0 + b_1y)$, where x and y are variables and a_i's and b_i's are constants.

The first step in the design procedure is the formation of a computation graph, variously known as a precedence graph or task graph. Each node in the computation graph represents an operation or subtask. The directed edges indicate the flow of information as well as the precedence among subtasks.

A task can have many different computation graphs for two reasons. The first is that there are many different algorithms to compute the same task. The second is that there are many ways to define subtasks of an algorithm. Choosing a subtask which is very small; that is, the operation at a node in the graph is very primitive, makes the graph very large.

For our task $z = (a_0 + a_1x + a_2x^2)(b_0 + b_1y)$, some of the ways to compute z are:

1. $z = a_0b_0 + (a_0b_1)y + a_1(b_0x) + (a_1b_1)(xy) + (a_2b_0)(x.x) + a_2(b_1y)(x\cdot x)$
2. $z = (a_0 + a_1x + a_2(x\cdot x))(b_0 + b_1y)$
3. $z = (a_0 + [a_1 + a_2x]x)(b_0 + b_1y)$.

For the same task some of the subtasks can be defined as follows:

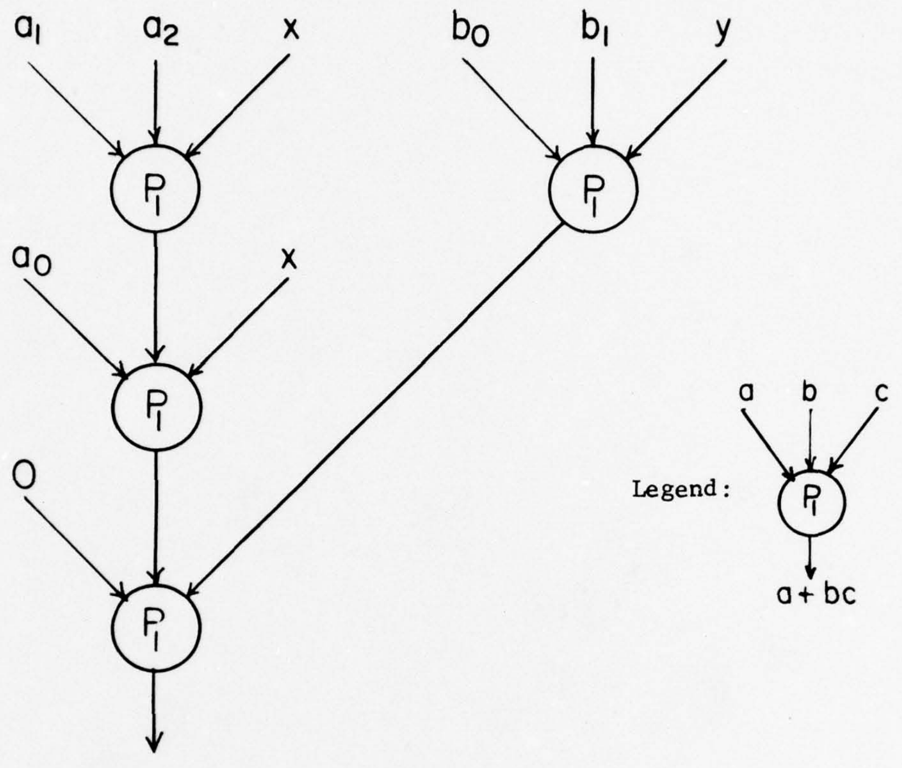
At a very low level: AND, OR, NOT operations.

At a moderately high level: Multiply and Add operations.

At a higher level: Operation which computes a polynomial of degree one; e.g., $P_1(a_0, a_1, x) = a_0 + a_1 x$.

As an illustration, Figure 5.2.1 shows the computation graph of expression $z = (a_0 + [a_1 + a_2 x] \cdot x)(b_0 + b_1 y)$ with subtask P_1 defined as $P_1(a,b,c) = a + bc$. Figure 5.2.2 shows the computation graph for the same expression but using smaller subtasks, namely, multiply and add.

The selection of subtasks can be done in the following way. Suppose the performance goal is one task every t seconds. Then, select a subtask which can be executed in approximately t/k seconds, where k is an integer. It determines the degree of maximum allowable sharing of any segment, that is, if the given performance goal is to be met, no more than k computation steps can be assigned to any one segment. A large k implies a finer definition of subtasks. k should be sufficiently large so that enough flexibility exists later in the design process for obtaining desired collision characteristics. However, it should not be so large that the resulting complexity of the reservation table makes the problem computationally unmanageable. Furthermore, if the design is to use components already available in the market then k should be chosen such that subtasks conform to operations available in these components. Of course if custom made components are to be used then this restriction does not apply.



$Z = (a_0 + a_1x + a_2x^2)(b_0 + b_1y)$

Figure 5.2.1. A computation graph of function Z.

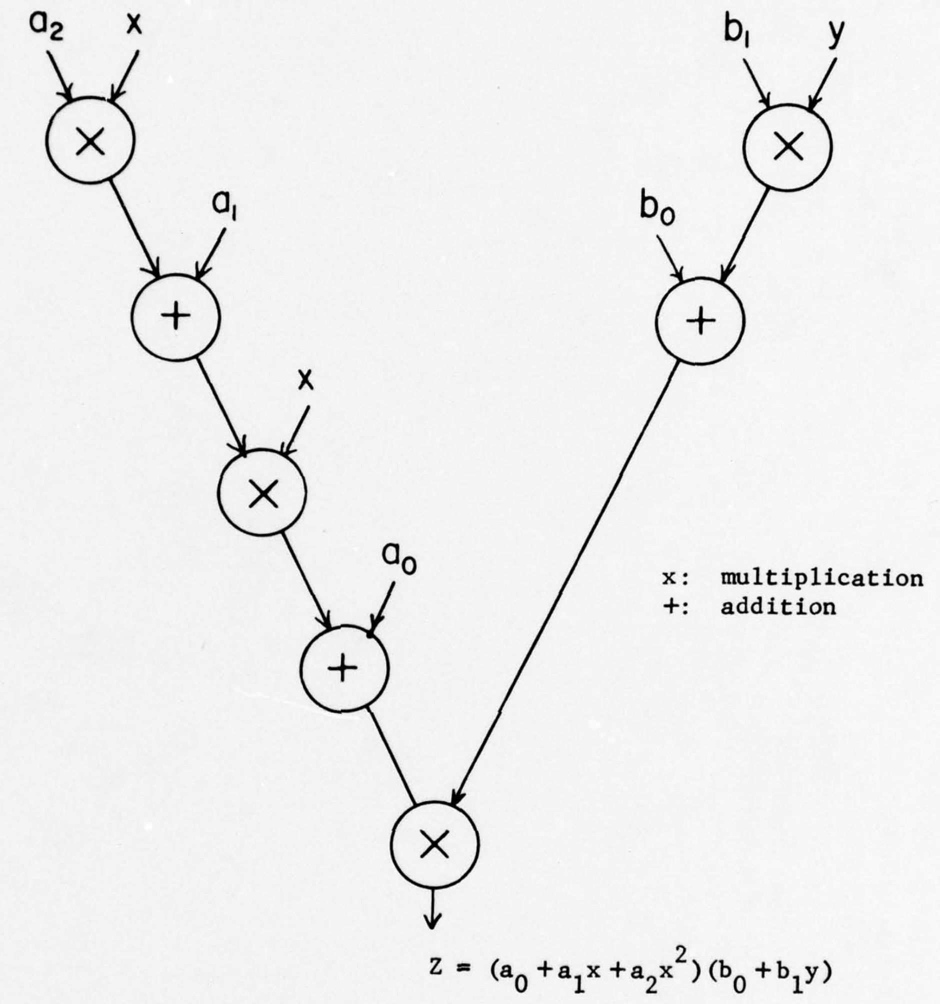


Figure 5.2.2. A computation graph of function Z.

Suppose in our example the performance goal is one task every few multiplication times then clearly the choice for subtasks is multiply and add operations. Both of these operations can also be performed by available circuit packages.

We have yet to say how we choose an optimum algorithm and what optimum means here. Clearly, the definition of an optimum depends on the goals of the design. Suppose execution delay is more critical than the cost, then we would choose an algorithm with the maximum parallelism, so that we can do as much computation as possible in parallel and thus reduce the execution time of each task. If the cost was of major importance then one would choose an algorithm with the fewest number of total subtasks or primitive operations, where the subtasks are simple and require little hardware.

It should be pointed out that selecting the proper subtasks and the proper algorithm are not really two sequential steps because we must have some idea about the algorithm before we can define the subtasks and the decomposition into subtasks and the resultant segment sharing determines cost and performance.

For our example let us choose an algorithm with the fewest subtasks, where subtasks are defined to be add and multiply operations. Figure 5.2.2 is a computation graph of such an algorithm which requires 3 adds and 4 multiplys.

The next step in the design procedure is the choice of functional units or segments to perform the subtasks of the computation graph. The choice of the right technology and devices is best left to the design engineer, because probably he is the one who is best informed of the existing state of the art in devices as well as various trade-offs between speed, cost, reliability, power consumption, layout, interconnections, etc. At this point it will suffice to say that the designer must strike a balance between the speeds of various segments because equalizing the utilization of all segments tends to give better performance per unit cost.

In our example, multiply is the bottleneck operation for two reasons. First, there are more multiply operations than add operations in the computation graph. Second, multiply is a somewhat slower process than add. Consider an add unit which adds two operands in 200 ns and a multiply unit which multiplies two operands in say, 4000 ns. To reduce this large speed gap we may use a fast multiply unit which takes say 2400 ns. Since multiply still remains the bottleneck we may choose a slow but inexpensive adder which takes say 1200 ns. One should not force a balance in speeds if no advantage results. Thus for example, the 2400 ns multiplier may be made twice as fast but only possible with a 500% increase in cost. Moreover, the resulting increased performance may be much over the established goal. Similarly for example making the 1200 ns adder half as fast may not reduce the cost significantly.

Let us assume for our illustration that the add segment takes 1 time unit and the multiply segment takes 2 time units, where a time unit is say 1200 ns. It is convenient to use such time units in the reservation table. In general, we take a time unit equal to the greatest common divisor (gcd) of the execution times of all segments. Since every segment must have a latch or output register, let us assume that execution time of each segment also includes the latch delay.

The time unit chosen in this manner determines the basic clock rate of the pipeline. Since, all execution times are expressed as some integral multiple of the time unit and since all data transfers occur synchronously at certain multiples of the basic time unit we choose the period of the clock equal to the basic time unit. Then a counter or a frequency divider can be used to generate control pulses at certain multiples of the clock period. Clearly, the larger these multiples the more expensive is the required frequency divider. These multiples can be made smaller if the clock period can be made larger. However, since the clock period is fixed by the gcd of execution times of the segments, the only way to increase the period is increase the gcd. This can be done by slightly adjusting the execution times of the segments. For example, if the adder execution time is 1201 ns and multiply 2400 ns then gcd (1201,2400) is 1. This implies a clock period equal to 1 ns. However, by taking the execution time of the multiply segment equal to 2402 ns, a minor adjustment of 2 ns, we get gcd(1201,2402) = 1201 ns implying a clock period of 1201 ns.

One more reason for making the gcd as large as possible by adjustment of segment times is the complexity of the reservation table. Again take the example of adder time = 1201 ns and multiply time = 2400 ns, the gcd being 1. Thus one add step will be represented by

1201 X's and one multiply step by 2400 X's. However if the add and multiply times were taken 1201 and 2402 ns respectively then the add step can be represented by only one X and the multiply step by 2 X's. Thus making the gcd large helps in simplifying the reservation table. Furthermore it is unlikely that any scheduling of the pipeline could exploit the 2 ns difference in mutiply time to obtain higher performance. Such minor adjustments in segment times will thus tend to simplify the pipeline and its scheduling without sacrificing much if any performance.

Once the basic time unit is determined, we are in a position to start assigning various subtasks to the segments and enter this information in a reservation table. We start out by forming a straight through pipeline by assigning a distinct segment for each node in the computation graph. Then we allow enough sharing of the segments so that the lower bound average latency of the pipeline is no greater than the target average latency. To illustrate the various points in the design process, we have used more than one target latency in the following.

The straight through reservation table for task Z of the tree type computation graph in Figure 5.2.2 is shown in Figure 5.2.3a. M and A in the figure stand for multiply and add respectively. The partial results obtained at the output of each segment are also shown on the right side of the table. It is convenient to retain some information on precedence among subtasks in the reservation table. Therefore, let us use an X for subfunction $(a_0 + a_1x + a_2x^2)$ and a Y for subfunction $(b_0 + b_1y)$ and rewrite the reservation table as in

	0	1	2	3	4	5	6	7		
S_0	Z	Z							M	a_2x
S_1			Z						A	$(a_2x + a_1)$
S_2				Z	Z				M	$(a_2x + a_1) \cdot x$
S_3						Z			A	$(a_2x + a_1) \cdot x + a_0$
S_4	Z	Z							M	b_1y
S_5			Z						A	$(b_1y + b_0)$
S_6							Z	Z	M	$(b_1y + b_0) \cdot [(a_2x + a_1) \cdot x + a_0]$

(a)

	0	1	2	3	4	5	6	7	
S_0	X	X							M
S_1			X						A
S_2				X	X				M
S_3						X			A
S_4	Y	Y							M
S_5			Y						A
S_6							Z	Z	M

(b)

Figure 5.2.3. Reservation tables of Figure 5.2.2.

Figure 5.2.3b, with the understanding that X and Y are independent of each other but both must precede Z.

Let us assume that the operands can be made available in the pipeline wherever and whenever they are needed by appropriate data routing. In our example a_i's and b_i's are constants and may be supplied internally exactly at those places where they are needed. The variable y is used only once starting at time instant 0. However, the variable x is used twice in the pipeline, once during time unit 0 and once during time unit 3. The operand x must be taken through delay segments so that it is available again at time instant 3. We have not shown any delay elements in the reservation table to keep it more readable. Delay elements are also omitted in all the reservation tables to follow. Once the reservation table is finalized and made allowable to an appropriate cycle we can assign the elemental delays to non-compute segments as was discussed in detail in Sections 2.4 and 2.9.

Suppose the performance goal is 1/8 tasks per unit time or equivalently, average initiation latency 8. Suppose it is required that the constant latency cycle (8) be allowable. The lower bound of Figure 5.2.3b is 2. If the lower bound need be no greater than 8 then we can share up to 8 usages of any segment. In Figure 5.2.4a all multiply operations are performed by segment S_0 and all add operations by S_1. The lower bound latency of Figure 5.2.4a is 8. This table can be made allowable to cycle (8) by inserting noncompute segments. Algorithm 2.7.1 can be used to accomplish this, but the constraints must be properly formed. In particular, the constraints should reflect

	0	1	2	3	4	5	6	7	8	
S_0	X	X	Y	Y	X	X		Z	Z	M
S_1			X		Y		X			A

(a)

	0	1	2	3	4	5	6	7	8	9	10	11	
S_0	X	X			X	X	Y	Y			Z	Z	M
S_1			X				X		Y				A

(b)

Figure 5.2.4. Pipeline with lower bound 8 made allowable with respect to cycle (8).

the independence between X and Y and dependence of Z on X and Y. Moreover, the indivisibility of the multiply step should also be taken into account. A minimum delay solution is given in Figure 5.2.4b. This solution has only one multiplier and one adder. However, this is not necessarily the minimum cost solution. As pointed out in Chapter 2, the computation of the actual cost is difficult, primarily due to lack of information on the data routing scheme. In this example, the cost factors consist of multipliers, adders, latches and multiplexers required for data routing and interconnections, as well as control logic. A designer must determine the cost of control, multiplexing, and interconnection from the data routing and scheduling schemes he intends to implement.

Next, suppose the performance goal is one task every 4 time units. Let us also suppose that execution delay is very critical and that any cycle is permissible if the delay can be kept small. We must bring down the lower bound of Figure 5.2.4a by duplicating or subdividing the multiply segment or using a faster multiply segment. Suppose we duplicate the multiply segment. There are three distinct ways in which four multiply operations can be assigned two each to two identical segments S_0 and S_2. These arrangements are shown in Figure 5.2.5 (based on Figure 5.2.3b). The X's and Y's are placed as soon as possible in these figures and are not placed for any particular collision characteristics. To obtain a minimum delay solution we must solve for all three tables and over all cycles with average latency 4, which is truly an impossibility! Therefore let us apply some heuristics.

	0	1	2	3	4	5	6	7	
S_0	X	X					Z	Z	M
S_1			X	Y		X			A
S_2	Y	Y		X	X				M

(a)

	0	1	2	3	4	5	6	7	
S_0	X	X		X	X				M
S_1			X	Y		X			A
S_2	Y	Y					Z	Z	M

(b)

	0	1	2	3	4	5	6	7	
S_0	X	X	Y	Y					M
S_1			X		Y	X			A
S_2				X	X		Z	Z	M

(c)

Figure 5.2.5. Three distinct assignments of four multiply operations to segments S_0 and S_2.

We commented in Section 2.11 that cycles with large periods are less likely to produce small delay solutions. So let us restrict our search for an optimum solution to the cycles with period less than or equal to 8, and of course with average latency equal to 4. All such distinct cycles are:

(4), (1,7), (2,6), (3,5).

By using the techniques of Section 2.9 we determine the largest allowable rows for these cycles. These can be done by forming the largest compatibility classes of these cycles. Such classes containing element 0, for each cycle are:

cycle (4): class $\{0,1,2,3\}$

cycle (1,7): class $\{0,2,4,6\}$

cycle (2,6): classes $\{0,1,4,5\},\{0,3,4,7\}$

cycle (3,5): class $\{0,2,4,6\}$.

The largest rows allowed by cycles (1,7) and (3,5) do not have any two consecutive elements. Thus a multiply operation, which uses two consecutive time units cannot be accommodated in the largest allowable row of these cycles. In other words, since multiply is assumed to be indivisible, the pipelines of Figure 5.2.5 cannot be made allowable with respect to cycles (1,7) and (3,5). The pipelines can be made allowable with respect to the rest of the cycles, namely, (4) and (2,6). A minimum delay solution for each pipeline of Figure 5.2.5 is given in Figures 5.2.6 and 5.2.7, respectively, for cycle (4) and cycle (2,6). The minimum delay solution among all these

	0	1	2	3	4	5	6	7	
S_0	X	X					Z	Z	M
S_1			X	Y		X			A
S_2		Y	Y	X	X				M

(a)

	0	1	2	3	4	5	6	7	8	9	10	
S_0	X	X					X	X				M
S_1			X			Y			X			A
S_2				Y	Y					Z	Z	M

(b)

	0	1	2	3	4	5	6	7	8	9	10	
S_0	X	X	Y	Y								M
S_1			X		Y	X						A
S_2				X	X					Z	Z	M

(c)

<u>Figure 5.2.6</u>. Pipelines of Figure 5.2.5 made allowable with respect to cycle (4).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
S_0	X	X											Z	Z	M
S_1			X	Y			X								A
S_2	Y	Y			X	X									M

(a)

	0	1	2	3	4	5	6	7	8	
S_0	X	X			X	X				M
S_1			X			Y	X			A
S_2				Y	Y			Z	Z	M

(b)

	0	1	2	3	4	5	6	7	8	
S_0	X	X			Y	Y				M
S_1			X			X	Y			A
S_2				X	X			Z	Z	M

(c)

<u>Figure 5.2.7</u>. Pipelines of Figure 5.2.5 made allowable with respect to cycle (2,6).

pipelines is that of Figure 5.2.6a in which the added execution delay is 0. Even though the added execution delay is zero, elemental delays must be inserted at various places for the proper flow of the operands. Solutions of Figures 5.2.7b and 5.2.7c are also attractive. For example, data routing might be simpler for the pipeline of Figure 5.2.7c, because segment S_0 has only one source, namely, the input part of the pipeline and only one sink, namely, segment S_1.

If the target was one task every two time units then we must replicate some segments to bring the lower bound to 2. A minimum delay pipeline allowed by cycle (2) is given in Figure 5.2.8. The added execution delay in this solution is zero.

To achieve a throughput of one task every time unit we need to divide the multiply segment so that the two time units of the multiply steps can be separated and assigned individually to two different segments. Ideally the division of a segment should divide the segment in equal parts, that is, the delay in each part is the same. Such a division is not always possible or practical for the following reasons.

1. If the original segment utilizes some internal feedback, for example, when the algorithm used by the segment is iterative such a physical partition of the segment cannot always be made. In particular if a feedback path lies across the intended dividing boundary then no such division can be made.
2. After a segment is divided each part requires a latch or output register for synchronization purposes. Since latches have to be

	0	1	2	3	4	5	6	7	
S_0	X	X							M
S_1			X			X			A
S_2				X	X				M
S_3	Y	Y							M
S_4			Y						A
S_5							Z	Z	M

<u>Figure 5.2.8</u>. Pipeline allowed by cycle (2).

inserted at division boundaries, a segment cannot be divided if it is available only as a single integrated circuit package.

3. Additional latches required after the division of a segment add delay to computation steps. This additional delay no longer permits the use of the original pipeline clock.

Some of the ways to circumvent the above problems are as follows.

1. Use a different algorithm for the segment so that internal feedback paths do not exist across a dividing boundary.
2. Slow down the pipeline clock to include the additional latch delays. This of course reduces the throughput from what it would have been if the clock was not altered.
3. For division use a faster segment to compensate for the additional latch delays and thus maintain the original clock timing.
4. Adjust the division boundary such that it does not lie across a feedback path or pass through an integrated circuit package. This of course may make uneven division of segment and as a result require some adjustment in clocking; thus degrading the performance from what it would have been with ideal division.
5. One more way to overcome the problem of feedback and integrated packages is to partially emulate the m-way division of a segment by multiplexing m copies of the segment. Thus for example division of a segment in two halves can be emulated by two copies of the original segment,a demultiplexer and a multiplexer. The demultiplexer switches the incoming tasks alternately between the two segments and the multiplexer remerges their outputs. The clock will have to

be adjusted due to the additional delay of multiplexing/demultiplexing. Such an arrangement emulates the successive usages of the parts of a divided segment.

In our example let us suppose that we have succeeded in dividing the multiply segments into two halves. The pipeline resulting from Figure 5.2.3b with lower bound one is given in Figure 5.2.9, where M_1 and M_2 are respectively the first and second half of the original multiply segment M.

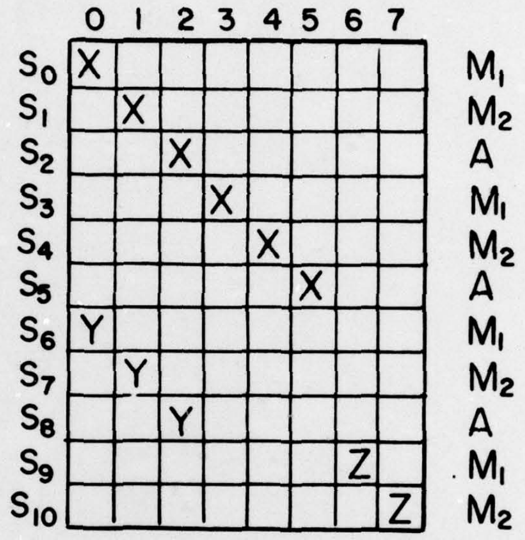
Any further increase in throughput requires further segment division. However, this is not an appropriate procedure to achieve more throughput. Remember that the choice of algorithm and definition of subtasks were not meant for a broad range of throughput. If the performance goal was very high then we should have started with a computation graph at a lower level by defining the subtasks on a finer scale.

So far we have considered only single function pipelines. Moreover, we restricted ourselves to using only fixed cycles for initiations. A single function pipeline can tolerate changes in the arrival pattern by a simple addition of an input buffer. The buffer can absorb the fluctuations in the arrivals and convert them into a desired initiation cycle. For these reasons in the design of single function pipelines there is no need for greedy control or internal buffers. Both these alternatives are likely to be less cost-effective than use of noncompute segments. However, in the case of multifunction pipelines noncompute segments are not always suitable for various

AD-A038 338   ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB   F/G 9/2
IMPROVING THE THROUGHPUT OF PIPELINES WITH DELAYS AND BUFFERS.(U)
OCT 76   J H PATEL   DAAB07-72-C-0259
UNCLASSIFIED   R-747   NL

3 OF 3
AD
A038338

END
DATE
FILMED
5-77



	0	1	2	3	4	5	6	7	
S_0	X								M_1
S_1		X							M_2
S_2			X						A
S_3				X					M_1
S_4					X				M_2
S_5						X			A
S_6	Y								M_1
S_7		Y							M_2
S_8			Y						A
S_9							Z		M_1
S_{10}								Z	M_2

Figure 5.2.9. Multiply segments divided to obtain a lower bound of 1.

reasons, such as the difficulty in finding a good initiation cycle and inability to process random arrivals. Therefore we shall need some alternatives to noncompute segments for the design of multifunction pipelines.

Multifunction pipelines: The same considerations apply in the formation of a computation graph as those for a single function pipeline. Next we form the reservation tables and use an appropriate control mechanism as follows.

If the arrivals are well defined so that an initiation cycle can be formed then we can use either noncompute segments or internal buffers. In either case the reservation table can be formed so as to meet or slightly exceed the required throughput. If the reservation table cannot be made allowable with the use of noncompute segments or if it can be made allowable only with a very high delay or cost then we should use internal buffers in the manner of Chapter 3. A simulation should be carried out to determine a good priority scheme and the actual buffer sizes required in the implementation.

If the arrivals are nondeterministic then we use either greedy control or internal buffers depending on the cost-effectiveness. If the cost of a single buffer is comparable to the cost of a segment, as is the case when the segment is made of only a few logic gates then the greedy control is likely to be far more cost-effective than the internal buffers as explained below.

If greedy control is chosen then the reservation table should be formed so that the potential throughput is considerably higher than the specified goal. Some of our simulations on a few specific examples have shown that a workload of 50% to 60% is processable with greedy control. Thus as an initial step the reservation table may be formed such that its potential throughput is about twice the required throughput. A simulation should then be carried out on this reservation table and throughput adjusted (by segment replication or division and sharing) accordingly until a satisfactory design is obtained. The description of the actual control mechanism for a greedy strategy can be found in [TH076].

If the costs of a segment and a buffer are comparable, then the cost of duplicating a segment is also comparable to that of adding a buffer. Since in our implementation of internal buffers, we must have at least one buffer per computation step, the cost of internal buffers may far exceed the cost of duplicating or even triplicating all segments. Thus even if greedy control requires the pipeline to have potentially twice or thrice the required throughput, it can be more cost-effective than the use of internal buffers.

When the cost of a single buffer is relatively low compared to that of a segment we may choose the internal buffers over greedy control. Here also, we must form the reservation table with a potentially higher throughput than the specified goal, since this choice reduces the average wait time of a task and also reduces the required buffer sizes as was indicated in Chapter 4. As pointed out in Section 4.4, it is advisable to carry out a simulation to determine a good priority and appropriate buffer sizes.

Designing pipelines with more capacity than required is not exclusive to multifunction pipelines with greedy control or internal buffers. Sometimes in the case of single function pipelines it helps to design a pipeline with more capability than required, that is, a pipeline with lower bound less than the required average latency. One reason for this is that segments may not be fully utilized and hence there is more freedom to move the usages (X's) of a segment so that the pipeline becomes allowable. In many instances, this increase in freedom results in a small delay solution. Another reason for designing a single function pipeline with higher capability than required is so that the cost of data routing might be low. This happens because the complexity of data routing increases faster than linearly with the increase in segment usages. Thus beyond a certain point the savings in the cost of segments might be more than offset by the increase in the cost of data routing. The quantification of the foregoing observations is very difficult in the general case and we shall not attempt it here. The following is a summary of the design methodology presented in this section.

5.3. Summary

1. Choose an algorithm (or algorithms in case of multifunction pipelines) for the task to be pipelined.
2. Define the subtasks based on the performance requirements, technology and availability of modules.

3. Form the computation graph of the algorithm using the subtasks defined in step 2.
4. Iterate steps 1 through 3 to obtain a computation graph with desired properties (e.g., shortest execution time or fewest number of subtasks or distinct subtasks, etc.).
5. Form a preliminary reservation table from the computation graph.

6a. [For all single function pipelines]. Modify the reservation table by segment replication or division so as to meet or slightly exceed the required throughput rate, that is, obtain a lower bound latency $\leq$ the required average latency. Form initiation cycles with the required average latency. Restrict the cycles to small periods. Form allowability and precedence constraints as described in Chapter 2. Use algorithm 2.7.1 to obtain all possible minimum delay solutions. Assign the elemental delays to noncompute segments as in Section 2.4 or 2.9. Choose a solution having the desired characteristics (e.g., minimum cost, minimum delay, etc.).

6b. [For multifunction pipelines with a well-defined arrival pattern]. Follow exactly the same procedure as in 6a. If the reservation table cannot be made allowable or if it can be made allowable only with very high cost and delay then use internal buffers. See Chapter 3 for specific details on internal buffers with periodic arrivals.

6c. [For multifunction pipelines with nondeterministic arrivals]. Modify the reservation table with segment replication or division such that the maximum possible throughput is considerably higher than the required throughput. Use greedy control if the cost of a segment is comparable to that of a buffer. Otherwise use internal buffers of appropriate size and priority using Chapter 4 as a guide. In either case simulations should be carried out before arriving at a final reservation table.

Even though the above procedure is given as a top-down design it is advisible to backtrack at several places for a better solution. Moreover, the cost tradeoff must be considered to compare segment replication and/or finer segmentation with the use of non-compute segments or internal buffers.

Chapter 6

CONCLUDING REMARKS

6.1. Conclusions and Summary of Results

We have shown that the theoretical maximum throughput of a pipeline can be achieved with the use of noncompute segments or internal buffers. We have allowed a considerable degree of freedom in the choice of an initiation cycle for achieving the maximum throughput. We have also shown that even with nondeterministic arrivals the use of internal buffers results in a near maximum throughput. The specific results can be summarized as follows.

The allowability characteristics of cycles and pipelines have been presented in considerable detail. It was established that modulo p equivalents of various quantities were sufficient when dealing with allowability, where p is the period of an initiation cycle. This resulted in a substantial reduction in the complexity of the theory. The concept of compatibility classes of a cycle was found to be very useful. From the compatibility classes one can determine the structure of all allowable pipelines of a given cycle.

We showed that under certain conditions a pipeline can be made allowable with respect to some cycle with insertion of delays. For a single function pipeline, it is always possible to add delays to it and make it allowable with respect to a constant latency cycle having a latency equal to or greater than the lower bound of the pipeline. We have developed systematic and effective procedures to insert delays in a pipeline to make it allowable for a given cycle.

We also investigated the effect of internal buffers in pipelines. Several different priority schemes were considered for resolving collisions. For priority, LIFO-global and any Precedence-Implied Priority it was shown that any cycle was executable with bounded queues and bounded wait time, if the cycle implied a workload no greater than 100%. Thus a great degree of freedom in the choice of a cycle was achieved with internal buffers. We also presented the upper bounds on the queue size and wait times for the above mentioned priorities. These bounds were found to be proportional to the number of initiations in a cycle.

When the arrivals were not assumed periodic we found by simulation that internal buffers were still very effective. A near maximum throughput was achieved in nearly all experiments using randomly structured pipelines. We also experimentally studied the dependence of wait time on buffer size, workload and priority.

Finally we presented a methodology to design pipelines with noncompute segments or internal buffers.

The results of this research are applicable to any pipeline which conforms to our assumed model. Many computations can be pipelined using our model. Notably, the arithmetic units of computers can be effectively designed and controlled using the techniques presented in this work. Many special purpose computations, such as Fast-Fourier-Transform, can also be effectively pipelined, while some existing pipelines can be modified to increase their throughput.

Finally, the pipelines modeled here are generally applicable beyond computational systems, for example to job shop and flow shop design and scheduling.

6.2. Suggestions for Further Research

Although we did cover a considerable number of properties of cycles, there is still need to refine some of the theory. As yet a simple method to test the perfectness of a cycle does not exist. Perfect cycles are useful since they allow 100% utilization of pipelines.

During the design of a pipeline in Chapter 5, we took a few cycles with small periods and made the pipeline allowable for each of these cycles and then we chose the best solution. We had to try several cycles, since we do not have any criterion to determine whether a particular cycle is going to have a good solution. Even the advice to use cycles with small periods is heuristic at best. Thus work remains to be done in finding an efficient algorithm to construct a "good" cycle, given a pipeline, so that when the pipeline is made allowable for that cycle, it requires the minimum possible added delay.

In the case of internally buffered pipelines with periodic initiations, the priorities FIFO-global, MPF and LWRF still remain to be solved. Two conjectures for Precedence-Implied Priority also need to be proven.

Some work needs to be done at the implementation level of the pipeline. Specifically, a systematic procedure is needed to establish a cost-effective data-routing scheme in the pipeline. Such a procedure is necessary to form a complete cost-model of the pipeline, since a considerable portion of the total cost of a complex pipeline is the cost of data-routing.

Almost every step of the design procedure of Chapter 5 could be made more rigorous and less heuristic. Each step, however, is a major research problem in itself.

LIST OF REFERENCES

AND67a D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM Journal of Res. and Dev., pp. 8-24, Jan. 1967.

AND67b S. F. Anderson, J. G. Earle, R. E. Goldschmidt and D. M. Powers, "The IBM System/360 Model 91: Floating-Point Execution Unit," IBM Journal of Res. and Dev., pp. 34-53, Jan. 1967.

BON69 P. Bonseigneur, "Description of the 7600 Computer System," Computer Group News, pp. 11-15, May 1969.

CHE71 T. C. Chen, "Parallelism, Pipelining and Computer Efficiency," Computer Design, pp. 69-74, Jan. 1971.

DAV71 E. S. Davidson, "The Design and Control of Pipelined Function Generators," Proc. 1971 Intl. IEEE Conf. on Systems, Networks, and Computers, Oaxtepec, Mexico, Jan. 1971.

DAV74 E. S. Davidson, "Scheduling for Pipelined Processors," Proc. 7th Annual Hawaii Intl. Conf. on Systems Sciences," pp. 58-60, Jan. 1974.

DAV75 E. S. Davidson, L. E. Shar, A. T. Thomas and J. H. Patel, "Effective Control for Pipelined Computers," Proc. Compcon Spring 75, pp. 181-184, Feb. 1975.

DOR67 N. M. Dor, "Guide to the Length of Buffer Storage Required for Random (Poisson) Input and Constant Output Rates," IEEE Trans. Comput., Vol. EC-16, pp. 683-684, Oct. 1967.

FLY72 M. J. Flynn, "Some Computer Organizations and their Effectiveness," IEEE Trans. Comput., Vol. C-21, pp. 948-960, Sept. 1972.

HAL72 T. G. Hallin and M. J. Flynn, "Pipelining of Arithmetic Functions," IEEE Trans. Comput., Vol. C-21, pp. 880-886, Aug. 1972.

HIN72 R. G. Hintz and D. P Tate, "Control Data STAR-100 Processor Design," Proc. Compcon Fall 72, pp. 1-4, Sept. 1972.

IBB72 R. N. Ibbett, "The MU5 Instruction Pipeline," The Computer Journal, Vol. 15, No. 1, pp. 42-50, Feb. 1972.

KNU73 D. E. Knuth, The Art of Computer Programming, Vol. 1, Fundamental Algorithms, 2nd ed., Addison-Wesley, Reading, Mass., 1973.

KNU75 D. E. Knuth, "Estimating the Efficiency of Backtrack Programs," Mathematics of Computation, Vol. 29, No. 129, pp. 121-136, Jan. 1975.

LAR73 A. G. Larson and E. S. Davidson, "Cost-Effective Design of Special-Purpose Processors: A Fast Fourier Transform Case Study," Proc. 11th Annual Allerton Conf. on Circuit and System Theory, pp. 547-557, Oct. 1973.

MAY75 W. Mayeda, "Two Methods for Improving Throughput of a Pipeline," Proc. 13th Annual Allerton Conf. on Circuit and System Theory, pp. 873-886, Oct. 1975.

PAR74 B. Parasuraman, "Pipelined Architectures for Microprocessors," Proc. Compcon Fall 74, pp. 225-228, 1974.

PHI70 G. Philokyprov and S. Tzafestas, "Buffer Size and Waiting-Time Calculations for Random Arrivals and Periodically Regular Service," Electronics Letters, Vol. 6, No. 18, pp. 588-590, Sept. 1970.

RED73 S. S. Reddi and C. V. Ramamoorthy, "A Scheduling Problem," Operational Res. Quarterly, Vol. 24, No. 3, pp. 441-446, Sept. 1973.

SHA72 L. E. Shar, "Design and Scheduling of Statically Configured Pipelines," Tech. Report No. 42, Digital Systems Lab., Stanford Univ., Sept. 1972.

SHA74 L. E. Shar and E. S. Davidson, "A Multimini-Processor System Implemented through Pipelining," Computer, Vol. 7, No. 2, pp. 42-50, Feb. 1974.

STE73 C. M. Stephenson, "Control of a Variable Configuration Pipelined Arithmetic Unit," Proc. 11th Annual Allerton Conf. on Circuit and System Theory, pp. 558-567, Oct. 1973.

STO73 H. S. Stone, Discrete Mathematical Structures and their Applications, Science Research Associates, Chicago, 1973.

THO74 A. T. Thomas and E. S. Davidson, "Scheduling of Multiconfigurable Pipelines," Proc. 12th Annual Allerton Conf. on Circuit and System Theory, pp. 658-670, Oct. 1974.

THO76 A. T. Thomas, "Scheduling of Multiconfigurable Pipelines," Tech. Report, Coordinated Science Lab., Univ. of Illinois-Urbana, 1976.

TOM67	R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal of Res. and Dev., pp. 25-33, Jan. 1967.
WAT72	W. J. Watson, "The TI ASC - A Highly Modular and Flexible Computer Architecture," Proc. FJCC 1972, pp. 221-228.
WIN73	A. K. Winslow, "Task Scheduling in a Class of Pipelined Systems," Report R-633, Coordinated Science Lab., Univ. of Illinois-Urbana, Nov. 1973.